NSDMiner: Automated Discovery of Network Service Dependencies

Arun Natarajan, Peng Ning, Yao Liu
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
Email: {anatara, pning, yliu20}@ncsu.edu

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

Steve E. Hutchinson
ICF Jacob and Sundstrom
Baltimore, MD 21202
Email: shutchinson@icfi.com

***Abstract*—Enterprise networks today host a wide variety of network services, which often depend on each other to provide and support network-based services and applications. Understanding such dependencies is essential for maintaining the well-being of an enterprise network and its applications, particularly in the presence of network attacks and failures. In a typical enterprise network, which is complex and dynamic in configuration, it is non-trivial to identify all these services and their dependencies. Several techniques have been developed to learn such dependencies automatically. However, they are either too complex to fine tune or cluttered with false positives and/or false negatives.**

In this paper, we propose a suite of novel techniques and develop a new tool named *NSDMiner* (which stands for Mining for Network Service Dependencies) to automatically discover the dependencies between network services from passively collected network traffic. NSDMiner is non-intrusive; it does not require any modification of existing software, or injection of network packets. More importantly, NSDMiner achieves higher accuracy than previous network-based approaches. Our experimental evaluation, which uses network traffic collected from our campus network, shows that NSDMiner outperforms the two best existing solutions significantly.

I. Introduction

Enterprise networks today host a wide variety of network services and applications. Many of these services and applications do not operate independently; they often depend on each other to provide and support network-based services and applications. For example, as illustrated in Figure 1, when a client accesses a web application, it usually first contacts a DNS server to resolve the IP address of the web server. Once contacted by the client, the Web Server further depends on an Authentication Server (e.g., Kerberos) to verify whether the client has the required privilege, and if yes, it relies on a Database Server for accessing data required to render the final output for the client. The dependencies between network-based services and applications are either hard-coded in configuration files or resolved dynamically.

Understanding the dependencies between network-based services is essential for maintaining the well-being of an enterprise network and its applications, particularly in the presence of network attacks and failures. For example, when a network-based application fails on the end host, it is important to know what network services are involved in this application and how they depend on each other to isolate and identify the faults. As another example, when a network is under malicious attacks, the knowledge of network services required for mission-critical applications, the dependency among them, and the availability of redundant services is invaluable information for prioritizing possible defense actions during the response to the attacks.

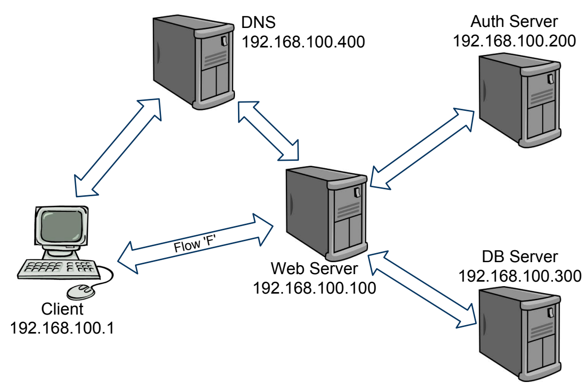


Fig. 1. An example of network services and their dependencies

A typical enterprise network is usually complex and dynamic in configuration. It is possible to understand some service dependencies through analyzing service configuration files. However, given the wide variety of network services, the lack of standard server configuration files, and the dynamic nature of enterprise networks, it is non-trivial, if not entirely impossible, to identify all network services and their dependencies through analyzing static service configuration files.

Several techniques have been developed to learn the network service dependencies automatically through analyzing the dynamic network traffic. In the following, we discuss these techniques and their limitations.

A. Related Work

Network-based Discovery of Service Dependencies: Several approaches have been proposed to automatically discover the dependencies among network services from network traffic. Various types of dependencies in distributed environments were identified in [13], and Leslie Graph was used as an abstraction to describe complex dependencies between network components [3]. Sherlock was developed to learn service dependencies based on co-occurrence of network traffic and was employed for fault localization [4]. eXpose uses a modified JMeasure computation on partitioned packet trace to learn dependencies [11]. Both Sherlock [4] and eXpose [11] require a window size parameter, whose choice influences significantly the detection accuracy. Orion was recently developed to use spike detection analysis in delay distribution of flow pairs to infer dependencies [8]. Moreover, fuzzy algorithms were used to build an inference engine for service dependencies [9].

All of the above approaches focus on analysis of network traffic, and thus do not require deployment of additional software on the end systems. However, they all share some common limitations. They all have high false positives and/or false negatives, and several of them are too complex to fine tune in order to produce meaningful results.

Host-based Discovery of Service Dependencies: Host-based approaches require deployment of additional software on individual hosts to discover the service dependencies involving these hosts. Macroscope uses an agent on each host to collect application and network data to infer the dependencies among network services and applications [14], achieving better accuracy with minimal false positives than previous network-based approaches. Host-based approaches employing middlewares have also been proposed to solve related problems such as fault correlation and failure management in distributed environments. Magpie [5] traces stand-alone events from operating systems and applications in a distributed environment to construct system behavior models for individual requests. pinpoint [7] tracks the path of each request as it passes through an distributed system for failure management. X-Trace [10] uses network events to reconstruct a users task tree.

All host-based solutions require running an agent on each host. This is not only intrusive, but also has a negative impacts on the performance of the end hosts. In typical enterprise or government networks, such an approach is in general difficult to deploy due to security and performance concerns.

Other Approaches: ADD uses active perturbation of applications to learn their dependencies [6]. However, this method requires the implementation details of the applications, and is time-consuming to implement considering the broad range of applications that exist today. Moreover, the approach in [12] uses configuration and software management repositories such as RPM and windows registry as information sources to build application-specific dependency models. Unfortunately, this approach misses dynamically resolved dependencies.

B. The NSDMiner Approach

In this paper, we develop a new technique called *NSDMiner* (which stands for *Mining Network Service Dependencies*) to automatically discover network service dependencies from passively observed network traffic.

Given two network-based services (or applications) A and B, we say A *depends on* B, denoted $A \rightarrow B$, if A is unable to complete its task without accessing B. The dependencies between network-based services can be classified into two categories: *local-remote dependency* and *remote-remote dependency* [8]. The dependency $A \rightarrow B$ is a local-remote dependency if A depends on a remote service B to provide a (local) service at A (e.g., a web server depends on a database server to render content of web pages). The dependency $A \rightarrow B$ is a remote-remote dependency if a client depends on a remote service B in order to access the remote service A (e.g., a browser depends on a DNS service to access a web server). As pointed out in [8], local-remote dependencies are commonly seen on servers, while remote-remote dependencies are commonly seen on clients.

In this paper, we focus on network-based effective and efficient discovery of local-remote dependencies, which are the common dependencies among servers. Our approach has several nice properties. First, it only uses passively observed network traffic as input, and thus does not rely on application behavior or configuration files to identify dependencies. Second, it is not restricted to only known services, and does not need any input about the existing network services and server infrastructure.

The contribution of this paper is two-fold. First, we develop NSDMiner, a novel technique for automated discovery of local-remote dependencies among network services from passively observed network traffic. Our solution is network-based, introducing minimal or no interference to the existing network infrastructure. Second, we implement a prototype of NSDMiner and perform extensive experiments to evaluate NSDMiner in a production network, with data collected over 26 servers hosting 40 instances of 13 different services. Our experimental comparison with Orion and Sherlock, which are the best among existing approaches for network-based service dependency discovery, shows that our method outperforms both Orion and Sherlock significantly.

The remainder of this paper is organized as follows. The next section discusses the intuition and design decisions involved in the development of NSDMiner. Section III presents the experimental evaluation of NSDMiner and the comparison with Orion and Sherlock. Section IV concludes this paper and briefly discusses some future research directions.

II. Design

A. Network Service & Service Dependency

We identify a *network service*, or just *service*, as a process running on a host that is accessed by applications or other services over the network to accomplish a specific task. The hosts offering these services are either preconfigured or resolved dynamically using a directory service or DNS. A service could run on a host at a well-known port (e.g., *ssh*, which runs at port 22) or a random port mapped by an endpoint mapper. Multiple services may run on the same host. Following [8], we identify a service by a triple *(ip, port, protocol)*.

As explained earlier, we represent a service dependency using the symbol "$\rightarrow$". For example, service A = *(Lip, Lport, Lprotocol)* depending on service B = *(Rip, Rport, Rprotocol)* is represented as *(Lip, Lport, Lproto)*$\rightarrow$*(Rip, Rport, Rproto)*, or $A \rightarrow B$.

Given a service dependency $A \rightarrow B$, we refer to A as the *downstream service* and B as the *upstream service*. Figure 1 illustrates an example of such dependencies, where a web service depends on an authentication service and a database service to serve its clients.

Sometimes, certain services are co-located at the same host. As a result, service requests between the services on the same host are not transmitted on the network. In this paper, we do not consider service dependencies that do not involve any network activity, but concentrate on the dependencies that are observable over the network.

B. Input

As discussed earlier, we discover the service dependencies based on passively observed network traffic. We assume that there are facilities in place to collect network activities of the services of interest. This monitoring could take place over a subnet or on individual hosts.

Our approach operates on network flows, including TCP and UDP flows. A TCP flow can be clearly identified when TCP is used as the transport layer protocol. A TCP flow starts with 3-way handshake (SYN, SYN-ACK, ACK) between a client and a server and terminates with a 4-way handshake (FIN, ACK, FIN, ACK) or RST packets. However, the notion of a UDP flow is a bit vague. When UDP is used as the transport layer protocol, there is no well-defined boundary for the start and the end of a conversation between a client and a server. In this paper, we consider a stream of consecutive UDP packets between two hosts as a UDP flow, if the time difference between any two consecutive packets is below a certain threshold.

We represent a TCP or UDP flow as a tuple of 7 attributes:

(StartTime, EndTime, SourceIP, SourcePort, Protocol, DestinationIP, DestinationPort),

where the attributes are self-explained from their names. For example, a flow may be represented as

(500, 502.3, 192.168.100.1, 2345, TCP, 192.168.100.100, 80),

indicating a TCP flow from source IP 192.168.100.1 at source port 2345 to destination IP 192.168.100.100 at port 80 from time 500 to time 502.3.

We refer to a flow as either *inbound* or *outbound* with respect to the machine. A flow is *inbound* when it is considered from the perspective of the destination, and *outbound* when it is considered from the perspective of the source. With respect to a host, we define the inbound flows as those initiated by the requests it receives from other hosts, and the outbound flows as the connections initiated by the host itself to other upstream services. Note that an outbound flow from a host is also an inbound flow to the upstream host. For example, in Figure 1, the flow 'F' is an outbound flow w.r.t. '192.168.100.1' and an inbound flow w.r.t. '192.168.100.100'.

C. Discovery of Services Dependencies

1) Observation: Our approach is based on the following observation about local-remote dependencies. Consider a service that depends on other upstream services to complete its own task. When this service receives an incoming request, it needs to initiate requests for the upstream services. In order to serve the original service request, all the upstream service requests initiated by this service take place while the original service request is still active. For example, when a client connects to a web server, the web server internally connects to an authentication server to authenticate the client and then connects to a database server for data access. All these outgoing connections from the web server are encompassed within the time frame of the inbound connection from the client. Figure 2 describes a timeline diagram showing the connections to and from a webserver.

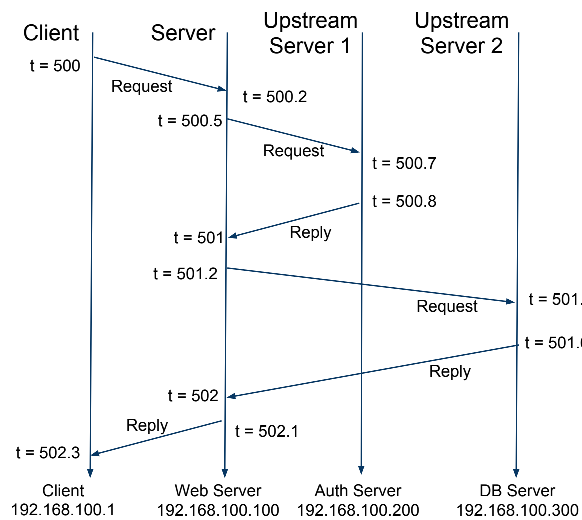


Fig. 2. A timeline of connections to and from a webserver

There are a few special cases in the way a downstream server depends on an upstream server. Even if a service depends on an upstream service, not every service request will trigger a connection to the upstream server. For example, not every request to a web server triggers a connection to the database server. Moreover, applications may optimize the connections between the servers using caching, such as cached domain name resolutions. In addition, some service may connect to upstream services even when it has not received any incoming requests due to the need of, for example, synchronization and authentication. We rely on a statistically significant number of samples to overcome these behaviors.

2) Algorithm: Based on the above observation we analyze the flow records for service dependencies. We track such dependencies using a metric *dweight*, which is the number of times a flow to the upstream service happened when the downstream service is invoked. A potential dependency $A \rightarrow B$ is tracked as *dweight*(A, B).

We process each flow record sorted in the order of their *StartTime*. On receiving a flow record, we consider it as an outbound flow record of the *SourceIP* and look back at all previous records to see if the current record is encompassed within the connection timeframe of any previous inbound flow record to the same *SourceIP*. If such an inbound flow record is found, the current outbound flow might be a flow that depends on the identified inbound flow, and can be considered as a candidate dependency. Thus, we increase *dweight* of the dependency tuple by one. The current flow is then stored, and is also considered as an inbound flow of the *DestinationIP* for the processing of future flow records.

Note that while processing the current record we check all previous records for dependency. Verifying all previous flow records is an expensive operation and the cost increases linearly over time. To reduce the memory footprint and for efficient processing, we remove all flow records that end before the start of the current flow, i.e. whose *EndTime* is less than the *StartTime* of current record. Thus, at any point during flow processing, we consider flow records that were active at that instance of time. This helps in achieving near constant processing time for every record.

Algorithm 1 Discover Service Dependencies

Input: FlowRecords (the set of flow records to be analyzed)
Output: Service dependencies
Steps:

```
PrevInbounds = {}
for all FL ⇐ FlowRecords do
  ServiceUsage(FL.DestServ) += 1
  for all PFL ⇐ PrevInbounds do
    if FL.time ⊂ PFL.time then
      Update dweight(PFL.DestServ → FL.DestServ)
    else if PFL.EndTime < FL.StartTime then
      Remove PFL from PrevInbounds
    end if
  end for
  Add FL to PrevInbounds
end for
for all tracked dependencies (A→B) do
  if dweight(A → B)/ServiceUsage(A) ≥ α then
    Output (A→B)
  end if
end for
```

The identified dependencies in the previous step could have false dependencies along with the true dependencies. The false dependencies are due to coincidental traffic that occur when a service is servicing a client. We rely on a large number of samples to reduce the false positives. We consider a dependency as true, if the ratio of its *dweight* to the number of times the service is accessed is at least α. In an identical case when a server depends on the dependency for every request it receives, α could be 1. But practically, the value is much lower because of caching, application behavior and coincidental traffic. We experiment the performance of our system with respect to this α in Section III.

3) Ambiguity of Dependencies: When a flow record is processed, it might be encompassed within the duration of more than one inbound flows. The outbound flow might be the result of either of the one inbound flows. Consider the three flow records F1, F2 and F3,

F1: (100.2, 105.1, 192.168.0.1, 6785, TCP, 192.168.100.100, 80)
F2: (101, 106.5, 192.168.0.2, 2348, TCP, 192.168.100.100, 22)
F3: (102, 102.8, 192.168.100.100, 8764, UDP, 192.168.100.200, 88)

The outgoing flow F3 is encompassed within the timeframe of both incoming flows F1 and F2. Thus, F1 and F2 are possibly equally dependent on the outgoing flow F3. Figure 3 illustrates such a situation.

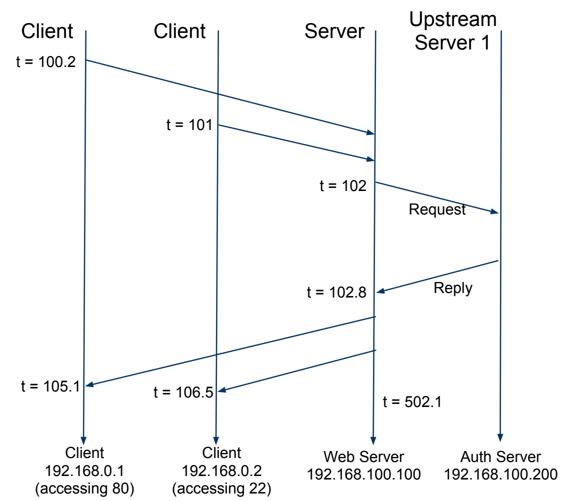


Fig. 3. An outbound flow encompassed within two incoming flows

To avoid making wrong decisions, we consider two different modes in addressing such ambiguous dependencies: the *shared* mode and the *exclusive* mode.

In the *shared* mode, we share the weight of current outbound flow among all candidate inbound flows. Specifically, if an outbound flow is encompassed within the time frame of n inbound flows, we update the *dweight* of each of these n tuples by $1/n$. We expect this approach to fairly distribute the weight among all potential dependencies in case of ambiguity. Though this solution updates the score evenly for the true dependency and the false ones caused by coincidental traffic, statistically significant number of samples will enable us to identify the true dependency. For example, while processing the above flow records our shared mode will update the *dweight* of both candidate dependencies by 0.5.

In the *exclusive* mode, we do not update the scores for any candidate dependency. Specifically, if an outbound flow is encompassed within the time frame of more than one inbound flow, we skip the current processing flow record and do not update the *dweight* of any candidate dependency. This approach is based on the intuition that though there are multiple flow records, we need only a few to exactly identify the dependencies. This approach is not quite effective when a system hosts multiple services and is used heavily by clients. In such cases, it is hard to find exclusive outbound flow traffic and we might miss some rare dependencies (i.e., the ones which occur at low frequencies). We will compare the *exclusive* and *shared* modes in Section III.

4) Service Aggregation: Many network infrastructures today deploy redundant services for load balancing and fault tolerance. In other words, there are clusters of servers providing identical services. When a service depends on such a service cluster, it may access any one of the identical upstream services. This choice could be made randomly at the machine or resolved dynamically through a different service. When the information of such a service cluster is known, we can use it to improve *NSDMiner*'s performance.

The services offered in such clusters usually have the same destination port and protocol but different destination addresses. The *dweight* of a candidate dependency on this service is distributed among the instances in the cluster based on the accessed pattern. To consider the dependency on this cluster of services as a unified one, we aggregate the *dweight* values of dependencies on the service instances in this cluster. Now the combined value better reflects the *dweight* of dependency on the service. Note that this knowledge of service clusters is not mandatory for our method, but could be used if available.

5) Long-running Flows: It is common to find long-running flows in any network trace, such as long SSH sessions or a Remote desktop connection, which could be kept alive for hours. These long-running flows pose a problem to our analysis, since any records processed within the timeframe when these flows are active will show a potential dependency relation. To remove such false positives we ignore all flows that are active over a long period of time.

D. Limitations

NSDMiner discovers service dependencies by analyzing the network flows. For the method to effectively identify a dependency, the services involved in the dependency have to be accessed over the network. If a service is rarely accessed by its clients, NSDMiner will have difficulty in identifying its dependencies. These limitations are not specific to NSDMiner, but shared by all network-based dependency discovery techniques. This limitation could be mitigated by collecting data over an extended period of time.

The accuracy of NSDMiner is still affected by a parameter α, particularly the false positive rate. Thus, NSDMiner is not fully free of user input. Nevertheless, parameter tuning is required by all existing approaches such as Orion [8] and Sherlock [4]. Moreover, as we will see in our evaluation in Section III-D3 even with a very low threshold NSDMiner still reports much lower false positives than the best existing solutions.

Finally, NSDMiner is not designed to discover remote-remote dependencies, as discussed earlier. Discovering remote-remote dependencies requires alternative techniques (e.g., Macroscope [14]).

III. Evaluation

We have implemented NSDMiner, including both the exclusive and the shared modes. In this section, we report the experimental evaluation of NSDMiner, including both the shared and the exclusive modes, and the comparison of NSDMiner with Orion and Sherlock, which are the most effective approaches among the previous network-based solutions.

To get realistic results, 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 monitoring facility used to collect the network traffic, and then present the experiment setup and the results.

A. Monitoring Facility

In this evaluation, we monitor the production network on the second floor of the building where the Department of Computer Science 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 multi-mode fiber, which facilitates communication with the campus network outside of the building.

To collect the network traffic, including the internal traffic, we set up SPAN sessions on all switches in the monitored network to capture the packets that pass through them. The switches are also configured to reduce the duplication in the SPAN sessions. For example, a packet that comes in and then out of a switch is reported only once. The switches send these packets to a collector switch through a dedicated 1G copper link, and the collector switch then aggregates all SPAN sessions and sends the packets through a dedicated 10G multi-mode fiber to a packet sniffer.

The packet sniffer is a Linux box with two 2.8GHz six-core Intel X5660 processors, 16GB RAM and 438GB HDD. The sniffer extracts the packet headers and exports it to a storage server through 10G single-mode fiber. The storage server has two 2.66 GHz six-core Intel X5650 processors with 24GB RAM and 1.3TB HDD running Gentoo Hardened Linux. Individual workstations were connected to the storage server for further processing of data. The entire monitoring infrastructure was encompassed within a private VLAN not accessible from any outside network to protect the data collected and ensure its security. Figure 4 shows the monitoring facility.

B. Experiment Setup

To capture network packets, we used *snort* [1] running in packet logging mode. To generate flows out of the captured packets, we modified *softflowd* [2], an open source software capable of tracking network flows. We reused most of the flow tracking code and added components to preserve client/server roles and export flows in our desired format. All flow records are sorted in the ascending order of the flow starting time.

We collected network flows for 46 days (05/10/2011 to 06/25/2011), getting around 378 million flow records. The production servers in the monitored network hosted a total of 40 known service instances. 23 servers ran Linux and 3 servers ran Windows. 18 servers hosted a single service, 5 hosted 2 services, and 3 hosted 3 or more services. These services had varied network loads and access patterns. The department web server and email server were heavily used,

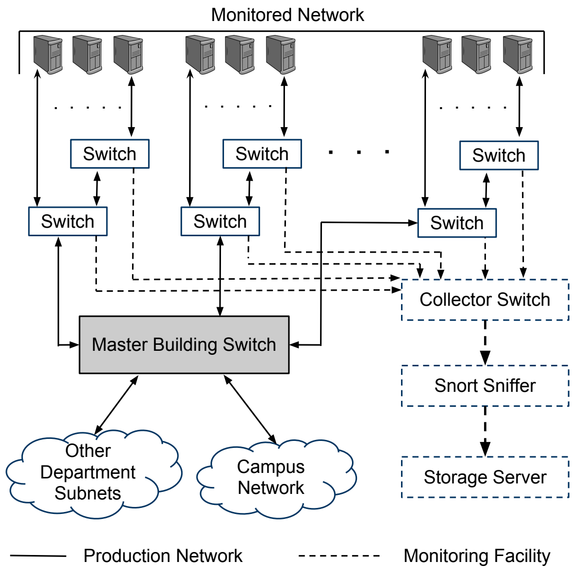


Fig. 4. The monitored network and monitoring facility

while services such as *ssh* were lightly used. 32 services were maintained by the departmental IT staff, while the remaining ones were managed by individual faculty members. Table I shows information about these servers.

TABLE I
OVERVIEW OF SERVERS

Server	OS	# Services	List of Services
a	Windows	1	svn
b	Windows	6	RPC, endpoint mapper, proxy DHCP, SMB, TFTP, DFS
c	Windows	6	RPC, endpoint mapper, DHCP, SMB, WDS, DFS
d	Linux	3	webservice (2), ssh
e	Linux	2	webservice, ssh
f	Linux	2	email, ssh
g	Linux	2	webservice, ssh
h	Linux	2	ssh, database
i	Linux	2	ssh, database
j-z	Linux	1	ssh

Besides the servers, an unknown number of computers were connected to the switches. We did not distinguish between servers and regular computers. Every connection initiated to any computer was monitored for dependencies.

The computers in the monitored network were often configured to use 3 server clusters outside of the monitored network, including DNS (6 servers), Active Directory (5 servers), and Kerberos (4 servers). These clusters were hosted and maintained by the campus network IT staff. With such heterogeneous mix, our monitored network reflects a typical setup in an Enterprise network.

NSDMiner is designed to run with minimal information from the user. None of these details is required to run NSDMiner.

During the dependency discovery process, we maintain the number of times every service is accessed over the network. A service has to be accessed a minimum number of times before its dependencies could be considered. In our experiments, we set this threshold as 50.

C. Ground Truth

Identifying the ground truth in the monitored network is a critical task. We established the ground truth with the help of our IT staff, since most of the services we studied were maintained by them. There was almost no documentation. Some of the services we monitored had well-known dependencies (e.g., DNS, authentication servers). However, there were also non-obvious dependencies that were not even known to our IT staff. Over the data collection period, we ran NSDMiner, Orion, and Sherlock over the collected data and repeatedly verified the results with our IT staff. In case of ambiguity, we referred to the configuration files of the services. After multiple iterations, we established the ground truth for evaluation.

Table II 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 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 II
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. In other words, our evaluation result on false negatives may be lower than the actual value, though we have high confidence in the false positive result. The reader is advised to keep this in mind while interpreting the experimental results.

D. Experimental Results

We ran NSDMiner in both shared and exclusive mode, collected the dependencies reported and compared it with the ground truth. Every reported dependency is classified as either a True Positive (TP) or False Positives (FP). The missed dependencies are counted as False Negatives (FN). We also ran experiments with the information about the server clusters to assess the impact of service aggregation, where the

TABLE III
DEPENDENCIES LEARNED

	NSDMiner ($\alpha = 0.5\%$)												Previous solutions, Best TP					
	exclusive			**shared**			**exclusive agg**			**shared agg**			**Orion agg**			**Sherlock**		
Service	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
webservice	8	4	1	8	9	1	9	3	0	9	5	0	7	13	2	4	47	5
email	3	0	0	3	1	0	2	0	1	2	1	1	2	2	1	2	2	1
ssh(ream-4)	10	1	0	10	1	0	10	1	0	10	1	0	10	1	0	10	27	0
ssh(ream-5)	40	10	11	41	14	10	40	5	11	41	6	10	27	3	24	27	70	24
svn	4	1	0	4	1	0	4	2	0	4	3	0	3	0	1	4	17	0
proxy DHCP	2	1	0	2	1	0	2	5	0	2	7	0	1	3	1	2	354	0
DHCP	1	6	0	1	6	0	1	3	0	1	3	0	1	2	0	1	46	0
endpoint mapper	3	6	3	6	43	0	3	6	3	6	24	0	2	0	4	6	67	0
WDS (RPC)	4	1	0	4	10	0	4	4	0	4	5	0	0	0	4	4	42	0
DFS replication (RPC)	7	0	1	8	4	0	7	2	1	8	9	0	4	1	4	3	142	5
SMB	9	3	1	9	7	1	10	4	0	10	9	0	9	4	1	10	263	0
TFTP	0	0	0	0	1	0	0	0	0	0	4	0	0	8	0	0	371	0
database	0	1	0	0	1	0	0	2	0	0	2	0	0	10	0	0	15	0
Invalid Services	0	30	0	0	34	0	0	24	0	0	32	0	0	14	0	0	2394	0
Total	91	64	17	96	133	12	92	61	16	97	111	11	66	61	42	73	3501	35

*dweight*s of dependencies on the services offered by clusters are aggregated. Throughout all the experiments, we configured the threshold $\alpha = 0.5\%$.

1) Initial Results: Table III summarizes the experimental results. The headings "shared" and "exclusive" refer to the basic modes without service aggregation, and "shared agg" and "exclusive agg" refer to the those with service aggregation.

The results show that NSDMiner is effective in identifying the service dependencies. We are able to identify up to 84-90% of dependencies with manageable false positives. Around 25-47% of the false positives were for possible (invalid) services that do not run on the servers managed by our IT staff. Strictly speaking, some of these may not be false positives; they may involve services on non-server machines managed by individual users. However, we do not have means to verify this. To be conservative, we consider them as false positives.

The remaining false positives and false negatives were primarily due to two reasons: (1) The first is the lack of active use of the service. This leads to insufficient samples to identify dependencies involving these services. Endpoint mapper is one such service, which has higher error rate than others due to lack of samples. (2) The second is rare dependency. Though the downstream service is heavily used, it rarely accesses the upstream service. The true negatives of *ssh* are of this type. The LDAP server is accessed only on a need-basis by the *ssh* service, which could be rare. NSDMiner missed this dependency on several services.

The experimental results in Table III also show that though server cluster information has marginal effect on the true positives, it is effective in reducing the false positive rate by up to 17% in the shared mode.

2) Comparison of Shared and Exclusive Modes: The experimental results indicate that the shared mode can get more true positives than the exclusive mode for some applications, but at the same time the share mode usually produces significantly higher false positives.

Regarding the better true positives, this is because the shared mode has less chance to miss the rare dependencies (than the exclusive mode) in case of flow co-occurrences (e.g., when an outbound flow is encompassed by two incoming flows). However, it has higher false positives because the shared mode distributes the *dweights* over multiple dependencies and continues tracking all the dependencies. For the same reason, the shared mode overall tracks more than double of the number of dependencies tracked by the exclusive mode. Figures 5(a) and 5(b) compare the number of services and the number of dependencies tracked in both modes w.r.t. the the actual numbers, respectively.

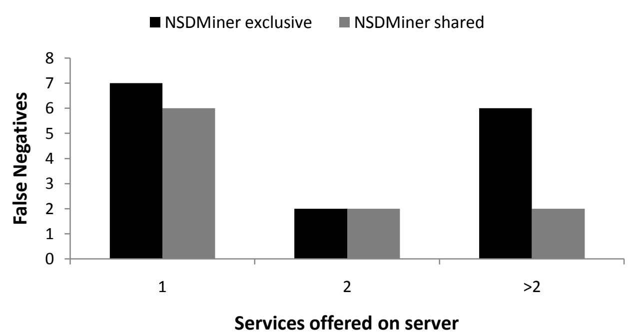
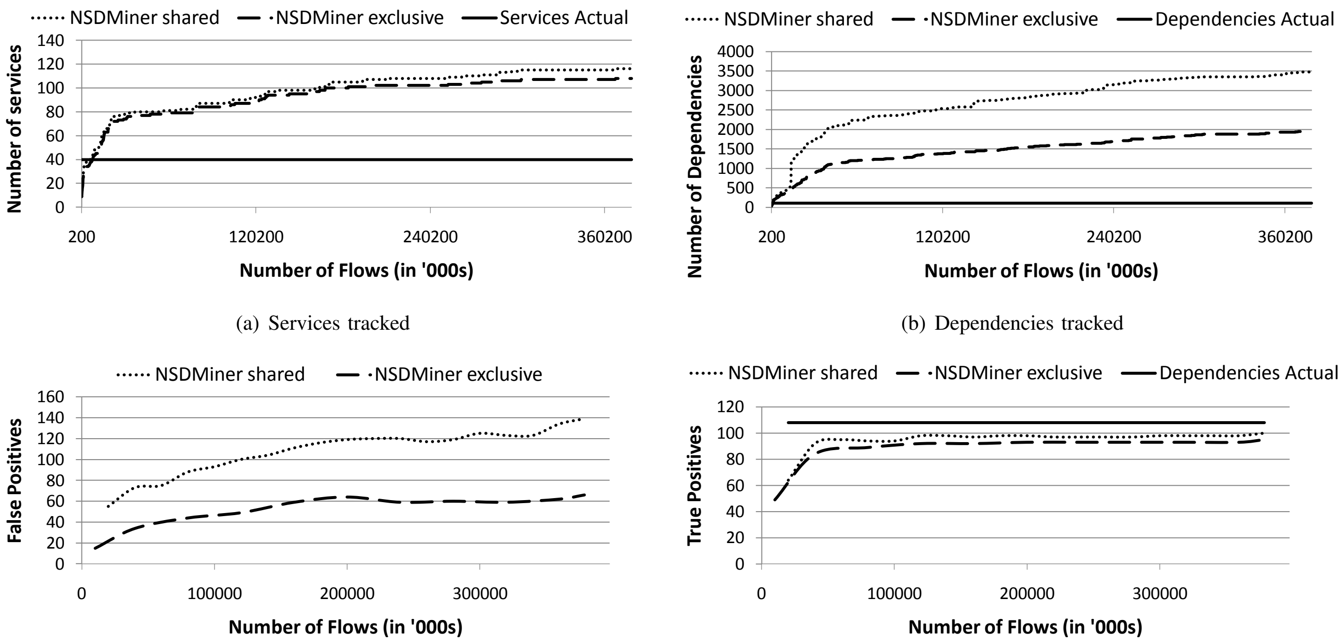


Fig. 6. False negative rate of servers by number of services hosted

Figure 6 compares the performance of the two modes based on the number of services running on the server. In the figure, the x-axis represents the servers offering only one, two, and more than two services, the y-axis represents the number of false negatives in each case. The number of false negatives of the exclusive mode is 2 times more than the shared mode on servers offering more than two service, while they are comparable on servers offering one or two services. This behavior is because the exclusive flows occur less frequently on systems hosting multiple services, and we missed some rare dependencies. Figure 5(c) compares the false positives and Figure 5(d) compares the true positives reported by both modes. The shared mode is able to identify marginally higher dependencies with almost twice as many number of false positives as the exclusive mode. Both modes converge to their results within almost one third of the flows processed, and



(a) Services tracked

(b) Dependencies tracked

(c) False positives ($\alpha = 0.5\%$)

(d) True positives ($\alpha = 0.5\%$)

Fig. 5. Shared v.s. Exclusive

as expected, false positives increases with further processing. This shows us that packets captured in 2–3 weeks are typically enough to learn the dependencies.

3) Comparison with Previous Approaches: To further evaluate our approach, we compared NSDMiner with two previous solutions, Orion [8] and Sherlock [4], which are the two more effective ones among the network-based approaches. We implemented both methods as described in their papers. In our implementation we tracked only local-remote dependencies and configured them with the same parameters used in [8] and [4]. For Sherlock we used 10ms time window and for Orion, we configured it with 300 bins, 10ms bin width, kaiser window of $\beta = 100$ and minimum 300 samples per dependency. Table III shows the results.

We also performed more experiments to compare these methods using receiver operating characteristic (ROC) curves. Specifically, we tune the parameter α in NSDMiner, the spike detection threshold in Orion, and the "chance co-occurrence" parameter in Sherlock to obtain the data points on the ROC curves. Figure 7 shows the result, where the x-axis and the y-axis represent the false positive rate and the detection rate (true positive rate), respectively.

To get a more complete picture of this comparison, we also added two data points for the aggregated server side dependencies (Exchange server and Sharepoint server) for Orion and Sherlock, using the evaluation results in [8]. However, we advise the reader that since the evaluation is heavily data dependent, comparing the data points obtained in [8] with our results is not necessarily meaningful.

Figure 7 shows that NSDMiner is able to identify more dependencies with less false positives compared with Orion and Sherlock. Sherlock shows a high number of false positives primarily because some services in our monitored network were co-hosted on the same servers and Sherlock mistook many co-occurring but independent service requests for service dependencies. As a result, most of its data points have 90% or higher false positive rates. Orion, though better than Sherlock, also missed dependencies. This might be due to varied loads on services and non-typical delays in accessing them. Figure 8 shows a similar comparison of NSDMiner and Orion with service aggregation. In both figures, NSDMiner consistently outperforms Orion and Sherlock in detecting dependencies with much lower false positive rate.

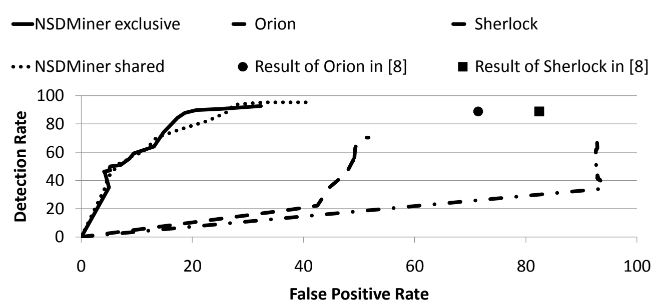


Fig. 7. ROC – NSDMiner v.s. Orion v.s. Sherlock

4) Effect of Threshold α: We now examine the impact of the threshold α on the effectiveness of our approach. Figures 9 and 10 plot false positive rate and false negative rate as a function of the threshold α in different modes. We can see that a large α can achieve a low false positive rate but result in a high false negative rate, whereas a small α leads to the opposite case. If the system goal is to balance both error rates, we may use the threshold that corresponds to the intersection of both error rate curves.

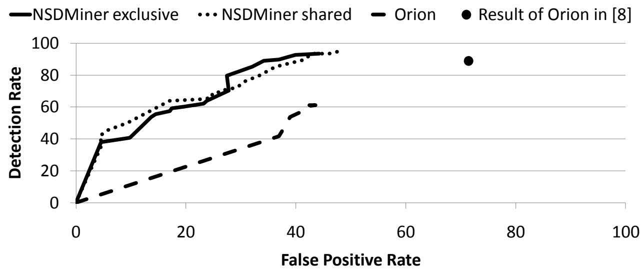


Fig. 8. ROC – NSDMiner v.s. Orion (with service aggregation)

In these figures, the balancing point ranges from 0.2% to 1% depending on the modes. Since we do not have this information without the ground truth, choosing α around 0.5% could be a good tradeoff. However, in practice, since the overall objective is to identify unknown service dependencies, discovering new dependencies (i.e., low false negative rate) is in general more important than dealing with false dependencies (i.e., high false positive rate). In other words, false positives can be filtered by examining the network services, but missing dependencies are difficult to identify. Thus, a small threshold is preferred if the number of false positives can be tolerated.

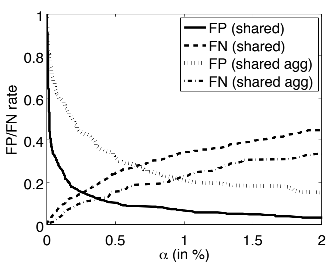


Fig. 9. NSDMiner shared

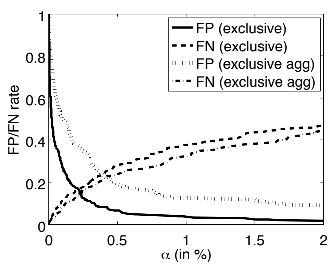


Fig. 10. NSDMiner exclusive

IV. Conclusion and Future Work

In this paper, we presented NSDMiner, a novel method to automatically discover service dependencies from passively observed network traffic. Our method is simple to tune and is resilient to varying network conditions. Our investigation indicates that the shared variation of NSDMiner is better for learning dependencies that involve hosts offering multiple services, and the exclusive variation is more suitable when most hosts offer services exclusively. NSDMiner can run with minimal user intervention. We evaluated NSDMiner with real-world network traffic collected from a production campus network. Our evaluation results show that NSDMiner can discover around 90% of the service dependencies with manageable false positives. Moreover, our evaluation also shows that NSDMiner outperforms existing solutions significantly.

Our future work is two-fold. First, we plan to extend the proposed method to discover service dependencies in real time, seeking techniques that can handle changing network configuration and server status. Second, besides the local-remote dependencies, we would like develop more effective techniques for remote-remote dependencies, particularly those that have much lower false positive rate than existing solutions.

V. Acknowledgments

This work is supported by the U.S. Army Research Office (ARO) under MURI grant W911NF-09-1-0525 and DURIP grant W911NF-11-1-0340. We would like to thank Dr. Xu Chen and Dr. Ming Zhang for their helpful inputs regarding the implementation of Orion. We would like to thank our department IT staff, Carlos A. Benavente, Vadim Kuznetsov, Trey Murdoch, and Jeremy Meeler, for their support and patience in sitting through long hours to manually collect the ground truth in our evaluation data set, and Neal McCorkle and William Brockelsby for their efforts in installing and maintaining the network monitoring facility, which allowed us to collect the experimental data.

References

[1] snort. *http://www.snort.org/* .

[2] softflowd. *http://code.google.com/p/softflowd/*.

[3] P. V. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering Dependencies for Network Management. In *5th ACM Workshop on Hot Topics in Networking (HotNets)*, November 2006.

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

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[6] A. Brown and D. Patterson. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment, 2001.

[7] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association.

[8] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, pages 117–130. USENIX Association, 2008.

[9] D. Dechouniotis, X. A. Dimitropoulos, A. Kind, and S. G. Denazis. Dependency detection using a fuzzy engine. In A. Clemm, L. Z. Granville, and R. Stadler, editors, *DSOM*, volume 4785 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 2007.

[10] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation*, pages 271–284.

[11] S. Kandula, R. Chandra, and D. Katabi. What's Going On? Learning Communication Rules in Edge Networks. In *ACM SIGCOMM*, Seattle, WA, August 2008.

[12] G. Kar, A. Keller, and S. Calo. Managing application services over service provider networks: architecture and dependency analysis. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 61 –74, 2000.

[13] A. Keller, U. Blumenthal, and G. Kar. Classification and computation of dependencies for distributed management. In *International Symposium on Computers and Communications*, pages 78–83, 2000.

[14] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft. Macroscope: end-point approach to networked application dependency discovery. In J. Liebeherr, G. Ventre, E. W. Biersack, and S. Keshav, editors, *CoNEXT*, pages 229–240. ACM, 2009.