Chapter 1

IDENTIFYING SUPERSPREADERS: RANKING SYSTEM OBJECT INSTANCE GRAPH

Rajani Mohan Suryavanshi, Xiaoyan Sun and Jun Dai

Abstract Defending today's enterprise network has become very challenging considering the increasing complexity and amount of cyber-attacks. It is critical to understand how an attack happens and how the intrusion propagates inside the network. System Object Instance Graph (SOIG) is a technique that can reveal the attack path at system level and thus help understand intrusion propagation. It captures dependencies among instances of system objects (files, processes, and sockets) and can be used to compute the infection probabilities of each instance. Attack paths are revealed by connecting instances with high infection probabilities. However, the SOIG can be very large and difficult to comprehend. Identifying the most important objects can aid human analysts' comprehension towards the intrusion propagation. Importance means not only high infection probabilities, but also large impact to other objects. Some objects are very impactful as they are heavily depended on by others and thus need to be scrutinized. If they are infected, they may generate huge negative impact by readily infecting others. Therefore, this paper proposes to rank the SOIG by extending the AssetRank algorithm. With the dependency ranks, security analysts can quickly identify the instances with highest impact. When combined with the infection probabilities, dependency ranks help analysts prioritize the objects that need most attention when time and resources are limited. The experiment results show that the proposed approach can successfully recognize the most impactful system object instances through the dependency ranks.

Keywords: System object instance graph, Dependency graph, AssetRank

1. Introduction

Enterprise networks usually contain very crucial assets and resources that can be of attackers' interest. Attacks to enterprise networks can

lead to security breaches, stolen data, substantial financial loss, disruption or even termination of businesses. The complete understanding of how attack happens is an indispensable component in digital forensics, and also crucial for recovery and better defense of the enterprise networks. However, since the complexity of attacks are also increasing, this makes detection and comprehension of attacks even more difficult. As the enterprise networks are often very complex in structure and size and may have several layers of protection such as firewalls, the attackers often need to launch multi-host, multi-stage cyber-attacks on these vulnerable networks [1]. The attacks usually start with an exploration of the target network by discovering the systems' configuration and passively monitoring their activities. The attackers will then try to exploit vulnerabilities on the systems, gains access to them and escalate the privileges. Attackers may also use some already corrupted systems as stepping-stones to further compromise other systems. Eventually the attackers fulfill their motives, such as embedding some malicious code into the system or stealing sensitive data.

To detect attacks and understand how the attacks take place and how the intrusion propagates inside the enterprise network, revealing the attack paths is often a very effective approach. This is made possible by two factors: 1) most networks are under constant surveillance by various security tools and techniques. Security sensors such as Snort [12], Tripwire [13], Wireshark [14], and Ntop [15] can be deployed across the network to capture security events. Alerts from various sensors on different hosts often reflect attackers' malicious activities, which can be chained together to help reveal attack paths. 2) system-level logs, such as system calls, can capture both the attackers' and legitimate users' activities in fine granularity with great details. Such information can significantly help understand how the intrusion has happened and propagated.

Some prior research works have been conducted in this direction. [1] proposed a solution to identify possible attack paths in an enterprise network. In this work, the network-wide attack context is represented using a superset graph called the System Object Dependency Graph. System objects include processes, files, and sockets. The dependency graph is constructed by analyzing the collected system calls from all the hosts in the network. Since the system calls are attack neutral, the superset graph thus captures activities from both attackers and normal users. Therefore, the attack paths can be identified from this superset graph by performing forward and backward tracking. Since this approach may result in path explosion problem, which means there may be too many suspicious attack paths identified, another study [2] pro-

posed ZePro, which uses a probabilistic approach to identifying attack paths. In ZePro, instead of using System Object Dependency Graph, it uses System Object Instance Graph (SOIG). SOIG is constructed similarly to System Object Dependency Graph except in SOIG, a node is not an object, but an instance of the object with a specific timestamp. Each instance is a new version of the object at different time points. This helps to capture infection statuses of an object at different time [2]. On top of SOIG, a Bayesian Network can be built so that evidence from the security sensor can be incorporated to calculate the infection probabilities of each instance. An instance with high infection probability means this instance may very likely be infected. By connecting the instances with high infection probabilities, an attack path can be revealed.

Although the above-mentioned approaches are effective to discover attack paths and understand the information propagation process, it may still be very challenging for security analysts to make good use of these paths due to the following reasons:

First, the size and complexity of SOIG are overwhelming. A 15-minute system call log contains 143,120 system calls, which involves 913 system objects, 39,361 object dependencies, 39840 instances, and 74598 instances dependencies [8]. Even after system call filtering and graph pruning, the size of SOIG is still very big. Please refer to Figure 7, Figure 9 and Figure 10 for parts of the SOIG.

Second, the revealed attack path may omit some very important objects that have significant impact on other objects. The attack paths only contain system object instances with the high infection probabilities, but these instances are not necessarily the ones that can propagate intrusion massively. In the SOIG, some instances have a large number of dependents or more important dependents, while others do not. Once they were infected, these impactful instances will generate much bigger negative impact by infecting more instances. Therefore, identifying and scrutinizing the most impactful instances is also critical to help understand the potential risks and how the intrusion may have been propagated.

Therefore, in this paper, we propose to rank the instances in the SOIG by extending the AssetRank algorithm [3]. Ranking SOIG will deal with the size as well as complexity of SOIG and help identify the most impactful instances. The AssetRank algorithm [3] is a modification of Google's PageRank [9] algorithm and can be used to compute the nodes' dependency ranks in a dependency graph. Hence it can be used to compute the ranks of object instances in SOIGs. The instances with high dependency ranks implies that they are heavily dependent on by other instances, and will need security analysts' more attention consid-

ering their potential impact. In addition, the dependency ranks can also be used together with the instances' infection probabilities computed by ZePro for better analysis of the instances. Instances with high infection probabilities may very likely be infected, but may not necessarily have massive impact on other instances, and vice versa. Therefore, based on the instances' dependency ranks and their infection probabilities, the instances can be classified into four categories: instances with both high dependency ranks and high infection probabilities, instances with high dependency ranks but low infection probabilities, instances with low dependency ranks but high infection probabilities, and instances with both low dependency ranks and low infection probabilities. This can greatly reduce the manual work of human analysts and help prioritize their work when time and resources are limited. Security analysts will need to firstly focus on the category of instances with high dependency ranks and high infection probabilities because these instances are the most dangerous and most impactful. When additional time and resources are available, they can then investigate instances in other categories, depending on their need, to analyze the instances with either high dependency ranks or high infection probabilities. To the best of our knowledge, this paper is the first work to identify the impactful system objects (including files, processes, and sockets) that may infect a large number of other objects at the low system level.

The paper is organized as follows: Section 2 introduces the background of this work. Section 3 describes the overview of the proposed approach. Section 4 provides the implementation details about generating the SOIG graphs and ranking the system object instances. Section 5 discusses the experiment, including the attack scenario, the SOIG generation and the result analysis. Section 6 provides a review of the related work. Section 7 concludes the entire paper.

2. Background

2.1 System Object Dependency Graph

System calls are usually the interface between the user applications and kernel operating systems. Applications need to call the system calls in order to get services provided by the operating systems. This applies to all applications no matter whether they are legitimate or malicious. Therefore, system calls are able to capture the activities from not only the legitimate users, but also the attackers. That's why system call logs are great resources to analyze and reveal attack activities.

In an enterprise network with multiple hosts, system calls can be collected on each individual host. Each system call involves one or more

system objects including processes, files, and sockets. Due to the actions in a system call, the system objects will have dependencies among each other. For example, when a process reads a file, the process depends on the file and receives data flow from the file. Therefore, by analyzing system calls, a dependency graph among system objects can be built [1].

A System Object Dependency Graph (SODG) is a directed graph comprising operating system objects as the nodes, and the dependencies among the nodes as edges. Every system call is broken down into three parts: source object, sink object and a dependency relation between the two objects. The objects are the nodes of the directed graph, and their dependency relations are represented by the directed edges between them. A socket, a process and a file are represented with a diamond, a rectangle, and a circle respectively. Below is a list of simplified system calls with timestamp $t1 - t3$.

```
t1: file 1 read by process A;
t2: process A creates process B;
t3: process B writes to file 1
```

The corresponding SODG is shown in Figure 1. The system call at $t1$ is parsed into a dependency relation "process A depends on file 1": $file\ 1 \rightarrow process\ A$. The system call at $t2$ is parsed into a dependency relation "process B depends on process A": $process\ A \rightarrow process\ B$. The system call at $t3$ is parsed into a dependency relation "file 1 depends on process B": $process\ B \rightarrow file\ 1$. The corresponding SODG in Figure 1 is built by connecting these dependency relations.

To build a network-wide SODG, a per-host SODG is constructed first and then they are concatenated. The concatenation is done if there exists a directed edge between two nodes on different hosts. This usually involves socket communication: the two nodes are two sockets and the directed edge represents the communication between the two sockets. For example, a *send* system call can indicate a socket $s1$ in machine X sending data to another socket $s2$ on machine Y. In this case, a dependency relation $socket\ s1 \rightarrow socket\ s2$ will be generated. Consequently, the per-host SODG for machine X and machine Y are concatenated into the network-wide SODG.

This network wide SODG is a superset graph and has a set of paths. It contains the attack paths if attack activities are involved. Starting from a triggering node (usually a suspicious object involved in a security alert), forward and backward tracking can be performed to reveal the attack paths.

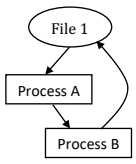

Figure 1. SODG.

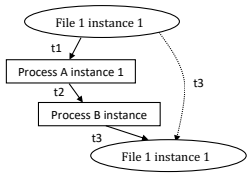

Figure 2. SOIG.

2.2 System Object Instance Graph

In order to calculate the infection probabilities of objects, Bayesian network (BN) is used in ZePro [8] to incorporate attack evidence (security alerts) and infer probabilities. However, it is not possible to directly build BN on top of SODG, as discussed in [8]: 1) SODG cannot preserve the time information of system calls and cause incorrect causality inference in the SODG-based BN; 2) BN is an *acyclic* model and does not allow cycles inherited from SODG (e.g. a cycle exists among File 1, Process A and Process B in Figure 1). Therefore, [2] proposed to use a new type of dependency graph, system object instance graph (SOIG) to build BN and calculate infection probabilities.

SOIG represents the dependencies between object instances. In SOIG, each node is an instance of the object at a specific timestamp. An object may be in a different state (infected or uninfected) at a different point of time. The state of an instance represents the state of an object at a specific time. Figure 2 shows the SOIG generated for the simplified example system call log. The label on each edge shows the time associated with the corresponding system call. The instance graph can reflect correct information flow as per timestamp. For example, Process B writes to File 1 at time instance $t3$. In SOIG, rather than connecting back to File 1, Process B points to the instance 1 of File 1. The creation of new instance of file 1 also breaks the cycle created in SODG by File 1, Process A and Process B.

Based on the SOIG, the Bayesian network (BN) can be built by assigning conditional probability tables to each node. The conditional probability tables define the causality relations among nodes, such as how the conditions of parent nodes can impact the condition of a child node. For example, in the Bayesian network based on Figure 2, the conditional probability table for node *Process A instance* 1 will define the probability of this node being infected based on the infection status of its parent node *File* 1 *instance* 1. The SOIG-based Bayesian network is able to incorporate the evidence gathered from security sensors such as Snort, Tripwire, etc. For example, if a file is observed to be infected,

its probability of being infected can be set into 1. This probability will affect the infection probabilities of the other nodes in the BN. After incorporating all the evidence, the infection probabilities of all nodes can be computed. Object instances with high infection probabilities have higher chances of being infected. Therefore, the infection probabilities can be analyzed together with the dependency ranks of these instances to recognize the instances that need security analysts' attention and further scrutinization.

3. Approach Overview

3.1 Overview of Dependency Rank Generation

Figure 3 show the overview of dependency rank generation. 1) Firstly, system call auditing should be performed on all the individual hosts. Such auditing will capture both the attackers' and legitimate users' activities. The OS-aware information such as file inode numbers, process IDs, as well as the timestamps will be preserved for the system calls. Filtering will be applied to remove system calls for some redundant and irrelevant objects. 2) After that, dependency relations will be extracted by parsing system calls. For example, a read system call indicating process P reading file F will be parsed to a dependency $F \rightarrow P$. 3) With the dependencies among system objects, a network-wide system object instance graph can be produced using the graph generation algorithm in [8]. 4) The network-wide SOIG can then be used as the foundation for achieving two goals, either to identify the attack paths (which is a subset of the SOIG), or to generate the dependency ranks of all instances. To reveal the attack paths, a Bayesian Networks should be built and the evidence collected from the security sensors, such as Snort, Wireshark, Tripwire, etc., will be incorporated to compute the infection probabilities of instances. The instances with high infection probabilities will form a path, which is the attack path. To generate the dependency ranks, the SOIG will be further parsed to convert the graph information such as node shape and dependency relations to the node types (AND, OR, SINK) required by AssetRank algorithm. The converted node type information is written to a .txt file. 5) Eventually the AssetRank algorithm is applied by taking the .txt file as input to compute the dependency ranks of all instances.

3.2 AssetRank of System Object Instances

PageRank Algorithm. Google's PageRank algorithm was developed to calculate the importance of webpages based on the number of

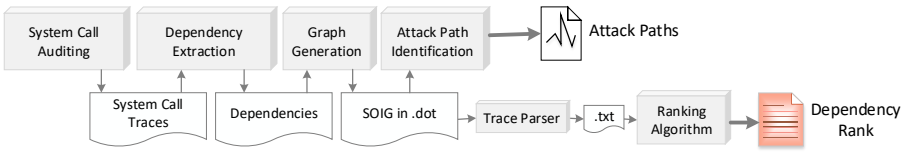

Figure 3. The Approach Overview

webpages linked to it. The method uses the webpages' link structure to calculate a rank and evaluate their importance. The calculations involved in a PageRank evaluation are completely general and can be applied to any scenario or domain that can be represented into a dependency graph [4].

In PageRank, a page has its own intrinsic value that indicates its independent and individual importance, and also gets extrinsic value from the pages linked to it. A page's rank can become higher if it gets more links. However, a page that is linked by a few important pages (i.e. pages with high intrinsic values) may have higher rank than a page linked by many unimportant pages.

AssetRank Algorithm. In this paper, we extended and applied the AssetRank algorithm [3] towards ranking the system object instances. AssetRank algorithm is a modification of the generalized model of the PageRank algorithm. It is applicable to graphs that models the dependency relationship among vertices, and can thus be used towards system object instance graphs, which describe dependencies among instances. In this paper, AssetRank is implemented to suit the semantics of system object instance graphs. It calculates the dependency rank value of the system object instances. The rank of each instance reflects how much other system object instances depend on it. The higher the rank, the more impact this instance may generate towards other instances.

In AssetRank, every vertex has a numeric value based on its importance in the network. The numeric value is formed of two elements, intrinsic value of the vertex itself and the value that a vertex receives from its dependents. The rank of a vertex is the total of the two elements [3]. Figure 4 depicts the value flow among the vertices. Vertices v_1 and v_2 depends on v_3; and vertices v_3 and v_4 depends on v_5. If v_1 depends on v_3, which means v_3 is important for v_1, so the value of v_1 will flow to v_3 and v_3's value is increased. The more nodes depend on v_3, the higher value v_3 has. That's why v_3 becomes darker. Similarly for v_5, some of v_3 and v_4's value flow to it and increased its value (but its value may be lower than v_3 because v_4 has a low value). Please be

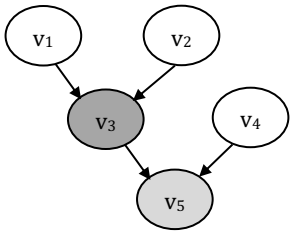

Figure 4. The Value Flow in AssetRank

noted that the arrows in Figure 4 denotes the value flow, rather than the dependency relations among vertices. The value flow actually has the opposite direction with dependencies in SOIG. In SOIG, v_1 depends on v_3 is denoted as $v_3 \rightarrow v_1$.

In addition, in the original PageRank algorithm [9], a vertex can be in logical relations of OR or AND. An OR vertex relation requires one of its preconditions (represented by out-neighbor) to be met. An AND vertex relation means all the preconditions present in the out-neighbors must be satisfied to reach the exploit or the vertex in the attack graph. SINK represents vertices which do not have out-neighbors. Considering the special causality relations among vertices in SOIG, when implementing AssetRank based on SOIG, a vertex will only be in OR relations. That means, if a process A depends on (e.g. reads) file B and C, as long as one of B or C is infected, process A will be infected.

Formally, a graph G can be defined as $G = (V, A, f, g, h)$ [3]. V, A, f, g and h respectively denote set of vertices, arcs, mapping of weights to vertices, weights to arcs and vertices to their type. The out-neighborhood and in-neighborhood of v are defined as:

$N^+(v) = \{w \in V\colon (v, w) \in A\}$ [3]

$N^-(v) = \{u \in V\colon (u, v) \in A\}$ [3]

The cardinality of a set is denoted as $|N|$. The total of all vertex weights should be 1. The total of all arc weights should be 1 if v is of type OR, and number of out-neighborhoods if v is an AND vertex. The sum need to be 0 if v is a sink.

$\Sigma_{w \in N^+(v)} g(v, w) = \begin{cases} 1, \text{ if } h(v) = OR \\ |N^+(v)|, \text{ if } h(v) = AND \end{cases}$ [3]

The value of vertex v contains its intrinsic value and the value that it gets from its dependents:

$x_v = \delta \Sigma_{u \in N^-(v)} g(u, v) x_u + (1 - \delta) f(v)$ [3]

δ denotes the damping factor, which defines the ratio between the value a vertex receives from its dependents and its intrinsic value.

The values for all vertices x_v can be put into a vector X. All the dependencies between the vertices are put into a weighed adjacency matrix D, and $D_{vu} = g(u, v)$ [3]. By applying Jacobi method of iteration to the sequence, vector X becomes:

$X = \delta DX + (1 - \delta)IV$ [3]

$X_t = \delta DX_{t-1} + (1 - \delta)IV$ [3]

The ranks of AND vertex and OR vertex is computed differently. When a vertex is OR type, the value of the vertex is split equally to all its dependents. But when the vertex is of AND type, the value of the vertex is replicated as it is to all its dependents [3]. By only considering the OR logic, the normalized equation becomes:

$X_{t'} = \delta DX_{t-1} + (1 - \delta)IV$ [3]

$X_t = \frac{1}{||X_{t'}||} X_{t'}$ [3]

The above sequence is supposed to be convergent (converges to 1 in this case). However, considering the large scale of SOIG, in rare cases where the sequence might not converge to 1 in a finite number of iterations, we have set a maximum number of iterations to 1,000, after which the algorithm will terminate if not converged to 1.

4. Implementation

The dependency rankings of system object instances are computed with the following processes. After collecting system calls, these system calls were parsed to generate the network-wide system object instance graphs, which is in the form of .dot file. DOT is a graph description language that can be used to describe graphs such as flowcharts and dependency trees [7]. The dot file for SOIG contains information such as node label (instance ID, instance name), shape (instance types, e.g. process, file, or socket), color (can be coded as infection probability, or dependency ranks) and edge directions (dependencies among instances). Graphviz software [6] is used to process and visualize the DOT files.

These network wide SOIGs are then fed into a parsing script written in Perl. The Eclipse IDE Neon version with EPIC plugin [5] is used for coding the parser. The parsing Script parses all the node information such as node shape and dependency relations with regular expressions, and converts the information to the node type (AND, OR, SINK) with the respective outward nodes[7]. If the shape is a box, then the node type is SINK. If the shape is an ellipse, then it is considered as OR. This node type information is stored in a text file which will be provided as input to the AssetRank java program.

For example, below is a simplified SOIG graph (whose graphical representation is shown in Figure 5) in .dot format:

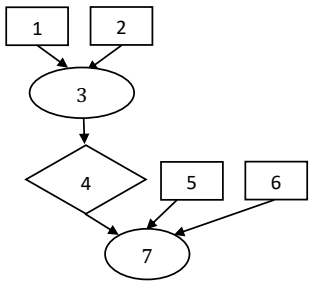

Figure 5. Graphical representation of Digraph ABC

```
Digraph ABC {
1[label = "1", shape = box, fillcolor = white];
2[label = "2", shape = box, fillcolor = white];
3[label = "3", shape = ellipse, fillcolor = white];
4[label = "4", shape = diamond, fillcolor = white];
5[label = "5", shape = box, fillcolor = white];
6[label = "6", shape = box, fillcolor = white];
7[label = "7", shape = ellipse, fillcolor = white];
1->3;
2->3;
3->4;
4->7;
5->7;
6->7;}
```

The Perl script will parse the above dot into a series of node types along with their dependents' information. The following output is stored in a text file and provided as input to the Java program:

```
SINK
SINK
AND, 1, 2
OR, 3
SINK
SINK
AND, 4, 5, 6
OR, 7
```

The ranking program will store the node information and their relative dependencies into an adjacency matrix. Each vertex's weight is computed by applying the AssetRank algorithm on the adjacency matrix. The result is the ranked SOIG which is a list of nodes with their respective ranks stored in an output csv file. These ranks are stored in sorted order with the highest ranked nodes stored first. If desired, this csv file, together with the original .dot file that contains the node information, can be used to visualize the ranks of instances on the SOIG. It will need to be input into another Perl script that parses the node ranks and assign the colors to instances based on their rank value. For example, the top 10% ranked nodes can be crimson color, and the nodes lying within 10% to 50% ranks can be in coral color, etc.

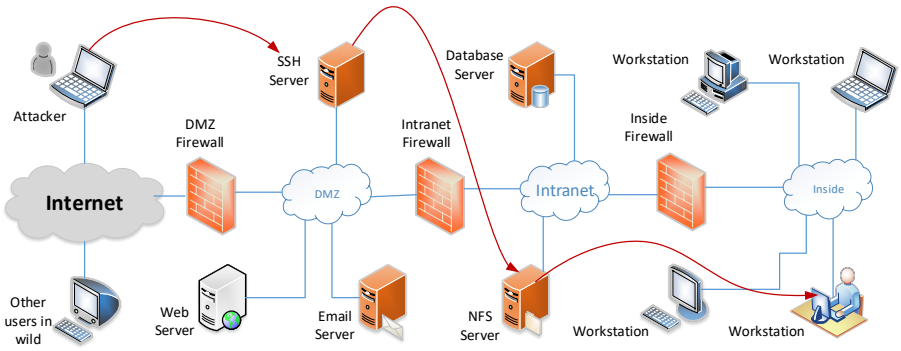

Figure 6. The Attack Scenario

5. Experiment

5.1 Attack Scenario

To analyze the dependency ranking of the system object instances together with their infection rate, we purposefully used the same attack scenario as in ZePro [2, 8]. In this way we can directly use the infection rates computed by ZePro. The attack scenario is shown in Figure 6 [2, 8]. The attacker firstly gains root privilege on a SSH server by exploiting a vulnerability CVE-2008-0166 [10], and can then access the NFS server using SSH as the stepping stone. Due to the wrong setup of export table on the NFS server, the attacker can upload a malicious executable (which contains a Trojan horse that exploits CVE-2009-2692 [11]) to a public directory on the file server. This malicious executable might be downloaded and installed by innocent users on the workstation. The attacker can then execute arbitrary code on the workstation. Security sensors Snort, Tripwire, Wireshark, and Ntop are deployed to capture security events that can be used as evidence of attacks.

5.2 Generated Network-wide System Object Instance Graph and Identified Zero-day Attack Path

With the attack scenario, ZePro system is used to generate a system object instance graph for the network. The SOIG usually captures both the attackers' and legitimate users' activities during the inspected time period, which is 15 minutes in this experiment. Since the graph is too large, it is not possible to include the entire graph in this pa-

Table 1. Infection Probabilities of Representative Instances.

Node	x4.2	x69.1	x142.25	x253.8	x254.7	x259.1	x260.1
Prob.	0.87	0.01	0.99	0.99	0.95	0.92	0.89
Node	x1007.6	x1008.5	x1017.1	x2005.1	x2006.3	x2083.1	x2082.2
Prob.	0.82	0.84	0.77	0.048	0.81	0.79	0.80
Node	x2086.5	x2114.2	x2147.2	x2158.5	x2397.21	x2383.3	x2429.14
Prob.	0.82	0.83	0.84	0.77	0.96	0.76	0.99

per. Therefore, Figure 7, Figure 9 and Figure 10 show different parts of the generated system object instance graph for each host involved in this attack. Readers are not expected to read the details of these SOIGs. However, to help readers understand what the nodes and edges look like, Figure 8 shows the magnification of the dotted circle part for Figure 7. In this graph, the instance of a socket, a process and a file are denoted with a diamond, a rectangle, and a circle respectively. For example, "$x198.1 : (192.168.202.2 : 36057)$" is a socket on the machine with IP 192.168.202.2 and port number 36057, and "$x199.1 : (6692 : 4935 : sshd)$" is a process with pid 6692, ppid 4935 and process name *sshd*.

For the generated SOIGs, ZePro system is able to compute the infection probability for each system object instance. That means, every node in the SOIG will get an infection probability value. Due to the large number of instances, we cannot include the infection probabilities for all nodes in this paper. Table 1 shows the infection probabilities of some representative instances.

Again, readers are not expected to understand the details of these graphs. However, these figures show how difficult it is to identify the system object instances that are most depended on by other instances and that can infect most system objects. If analyzed together with the infection probabilities, the dependency ranking can help analysts quickly identify the most important objects that needs their attention.

Since SOIG contains the attackers' malicious activities, the instances with high infection probabilities can be further identified and revealed. These instances form the attack path, or the zero-day attack path if zero-day attacks are involved [2, 8]. To make this paper self-contained, we also included the identified zero-day attack path, shown in Figure 11, to be used for result analysis.

5.3 Results

Based on the generated SOIG shown in Figure 7, Figure 9 and Figure 10, we computed the dependency ranks of all the 1853 nodes on

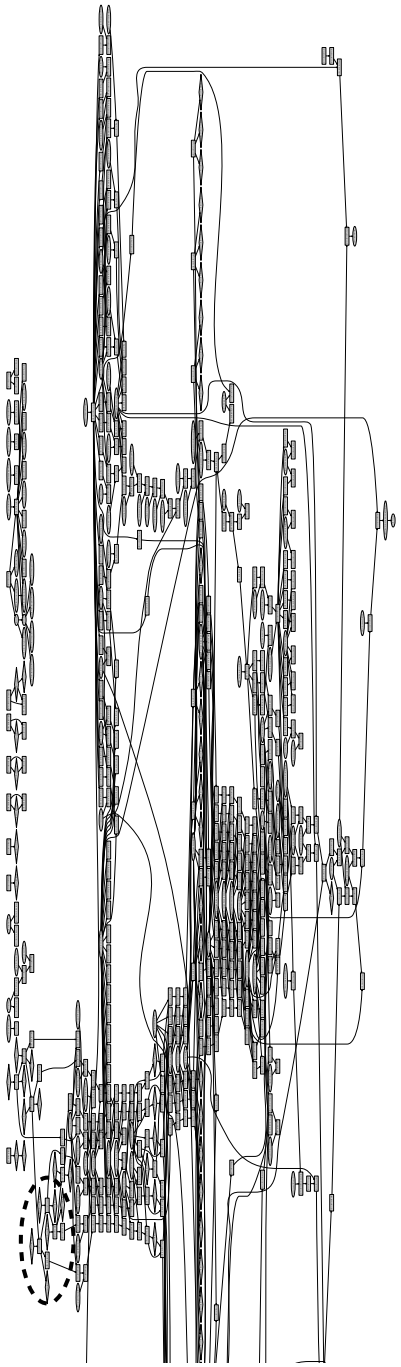

Figure 7. A snippet of the Generated System Object Instance Graph for SSH server

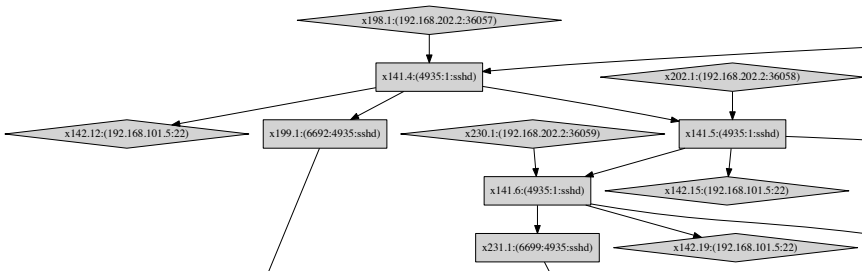

Figure 8. The Magnification of the Dotted Circle Part in Figure 7

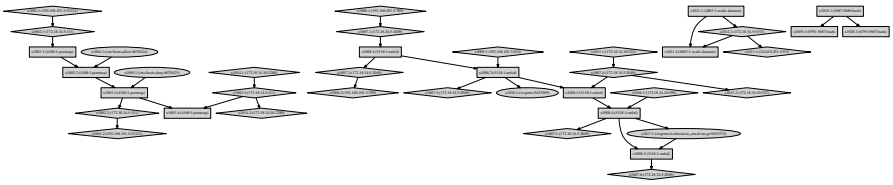

Figure 9. The Generated System Object Instance Graph for NFS server

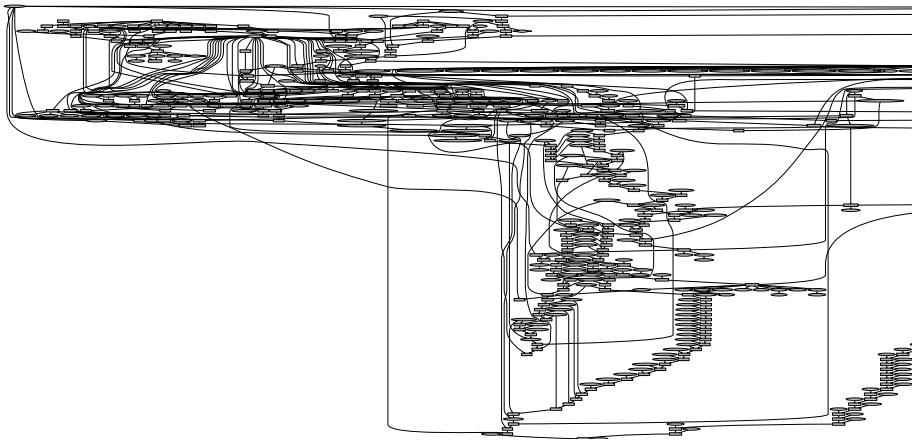

Figure 10. A snippet of the Generated System Object Instance Graph for the workstation

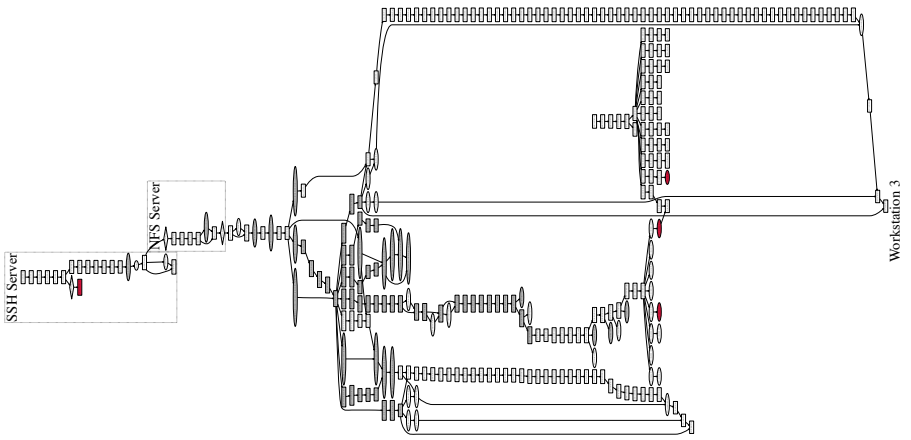

Figure 11. The Identified Zero-day Attack Path [2, 8]

the network wide SOIG. We kept the damping factor as 0.85, which is the standard value maintained in all experiments used in the PageRank algorithm. We treated all the nodes in the SOIG with equal importance at the beginning of the algorithm and then calculated their ranks iteratively till the equation converged to 1 or reached the maximum iteration count (which is set to 1,000 in this experiment).

Due to limited space, Table 2 shows the rank values of the top 50 nodes that have the highest dependency ranks on the SOIG. As per our observation, the instances with a greater number of out-going edges tend to have a higher rank value. This is because, as per AssetRank algorithm, an instance's AssetRank is made up of two parts: the intrinsic value, and the value it gets from its dependents. The vertex with multiple dependents tends to have a higher AssetRank. Analysis can then be conducted based on both the instances' rank values and their infection probabilities computed in [2, 8]. Scrutinizing Table 2 and Figure 11 can reveal that instances with the highest dependency ranks may not have the highest infection probabilities and vice versa.

Generally, we can categorize all the system object instances into the four categories below:

Instances with high dependency ranks as well as high infection probabilities. The instances in this category deserve most attention from the security analysts. High dependency ranks mean that the node has many dependents connected to it throughout the network. If the node's infection probability is also high, the attack will propagate through the network very easily. For example, some instances appear

Table 2. Dependency Ranks of Top 50 Instances on SOIG.

Rank	Instance	Object	Rank Value
1	2086.5	6763:6719:tar	0.0005927777894456
2	290.1	6729:6724:savelog	0.0005830315634006
3	2005.1	/etc/locale.alias	0.0005746923615380
4	69.1	/etc/nsswitch.conf:8798393	0.0005713089890883
5	25.1	5908:5897:apt	0.0005639606663220
6	2023.2	2816:1:udevd	0.0005630498216020
7	2147.2	6783:6781:exploit.sh	0.0005597664307222
8	101.4	6620:6604:cupsys	0.0005597639076630
9	2124.1	6285:5798:bash	0.0005597588152304
10	2493.12	6815:6813:bash	0.0005581044873810
11	263.2	6709:5897:run-parts	0.0005564425363920
12	223.2	2774:1:udevd	0.0005564349986352
13	2091.1	/etc/nsswitch.conf:1491129	0.0005532248168440
14	2429.14	6811:6803:useradd	0.0005531413877282
15	45.1	/etc/localtime:8798371	0.0005523218315349
16	188.1	/proc/sys/kernel/ngroups_max:12635	0.0005515249821289
17	2026.2	/:2	0.0005514693370436
18	52.1	5897:5896:run-parts	0.0005481961040248
19	183.1	/etc/pam.d/common-account:8799591	0.0005481935267130
20	78.2	6604:6603:sh	0.0005481833960051
21	182.1	/etc/pam.d/common-auth:8799592	0.0005481783462166
22	186.1	/etc/pam.d/other	0.0005481783345730
23	181.1	/etc/pam.d/sshd:8800291	0.0005481783345611
24	184.1	/etc/pam.d/common-session:8799594	0.0005481783345492
25	185.1	/etc/pam.d/common-password:8799593	0.0005481783345434
26	62.1	/proc/filesystems:4026531844	0.0005481783345433
27	2076.1	/bin/ls:532536	0.0005481783345137
28	2074.1	6719:6718:bash	0.0005481758038124
29	189.1	/home/jun/.ssh/authorized_keys:4636779	0.0005473713971032
30	4.1	6560:6559:mount.nfs	0.0005473562399357
31	2375.1	6796:6793:collect2	0.0005473473551626
32	2390.1	6801:6798:collect2	0.0005473473551626
33	226.2	/:2	0.0005465138528115
34	223.1	2774:1:udevd	0.0005457474421339
35	2023.1	2816:1:udevd	0.0005457335034186
36	144.1	/usr/sbin/sshd:9462668	0.0005457094577282
37	153.7	6687:4935:sshd	0.0005457043885236
38	2006.2	6737:6736:mount	0.0005457031115441
39	152.1	/etc/hosts.deny:8798361	0.0005457018810209
40	2081.1	/proc/filesystems:4026531844	0.0005456993367521
41	2308.3	6793:6783:cc	0.0005456968001993
42	2383.3	6798:6783:cc	0.0005456968001993
43	123.1	/etc/hosts:8798410	0.0005456967964019
44	301.1	/bin/mv:966724	0.0005456967963309
45	194.16	6690:4935:sshd	0.0005456942656000
46	162.1	6679:4935:sshd	0.0005456942578573
47	2397.1	6803:6783:wunderbar_empor	0.0005456917348751
48	231.16	6699:4935:sshd	0.0005456892041560
49	160.1	6680:4935:sshd	0.0005456892041382
50	164.1	6682:4935:sshd	0.0005456892041382

Table 3. Dependency Ranks of Top 50 Instances on the Identified Zero-day Attack Path

Rank	Instance	Object	Rank Value
1	2086.5	6763:6719:tar	0.0005927777894456
2	2147.2	6783:6781:exploit.sh	0.0005597664307222
3	2493.12	6815:6813:bash	0.0005581044873810
4	2429.14	6811:6803:useradd	0.0005531413877282
5	4.1	6560:6559:mount.nfs	0.0005473562399357
6	2006.2	6737:6736:mount	0.0005457031115441
7	2383.3	6798:6783:cc	0.0005456968001993
8	2308.3	6793:6783:cc	0.0005456968001993
9	2397.101	wunderbar_empor	0.0005456917348751
10	2429.7	6811:6803:useradd	0.0005440424297800
11	1008.4	5118:1:unfsd	0.0005440411760642
12	2397.8	wunderbar_empor	0.0005440411760287
13	2397.11	wunderbar_empor	0.0005440399068096
14	1008.2	5118:1:unfsd	0.0005440373838006
15	253.8	6706:6703:sshd	0.0005432279583837
16	2144.4	6781:6285:bash	0.0005432128164641
17	2086.4	6763:6719:tar	0.0005424676923594
18	1008.1	5118:1:unfsd	0.0005423956554335
19	2493.11	6815:6813:bash	0.0005423906095075
20	2429.9	6811:6803:useradd	0.0005423906094453
21	2429.101	6811:6803:useradd	0.0005423906055799
22	2429.11	6811:6803:useradd	0.0005423906055798
23	2397.201	wunderbar_empor	0.0005423880748490
24	2114.2	...wunderbar_emporium/pwnkernel.c	0.0005423868133430
25	2397.14	wunderbar_empor	0.0005423855518548
26	2397.9	wunderbar_empor	0.0005423855479953
27	2397.19	wunderbar_empor	0.0005423855479894
28	10.1	/etc/mtab:8798397	0.0005415710300122
29	1007.1	172.18.34.5:2049	0.0005415659839528
30	259.1	/mnt/workstation_attack.tar.gz:9453574	0.0005415634378076
31	2114.1	...wunderbar_emporium/pwnkernel.c:	0.0005415634281084
32	2082.2	/home/..attack.tar.gz	0.0005415622864635
33	1008.5	5118:1:unfsd	0.0005415609225679
34	2397.21	wunderbar_empor	0.0005415608993991
35	2006.3	6737:6736:mount	0.0005415608993162
36	2503.1	6818:6815:bash	0.0005415608993102
37	2385.301	6799:6798:cc1	0.0005415608993102
38	2385.3	6799:6798:cc1	0.0005415608993102
39	2311.401	6794:6793:cc1	0.0005415608993102
40	2152.3	...wunderbar_emporium/pwnkernel1.c	0.0005415608993102
41	2452.1	/etc/shadow+:1493108	0.0005415608954389
42	2388.2	/tmp/ccUZcd3t.o:2984227	0.0005415608954389
43	2372.2	/tmp/ccfRR34r.o:2984223	0.0005415608954389
44	2527.3	6830:6815:bash	0.0005415583685793
45	2458.1	/etc/gshadow+:1493110	0.0005415583685793
46	2455.1	/etc/group+:1493109	0.0005415583685793
47	2449.1	/etc/passwd+:1493107	0.0005415583685793
48	2443.1	/etc/gshadow.6811:1493106	0.0005415583685793
49	2440.1	/etc/group.6811:1493105	0.0005415583685793
50	2437.1	/etc/shadow.6811:1493104	0.0005415583685793

in both Table 2 and Figure 11, such as Instance 2086.5 (process *tar*) and Instance 2147.2 (process *exploit.sh*). Both instances are critical for the success of the attack. As shown in Table 1, the infection probability of Instance 2086.5 is 0.82, which indicates it is very likely infected; meanwhile, it is ranked No.1 in Table 2, which means it is dependent on by a lot of other nodes and may propagate infection to them very easily. Hence, if possible, security analysts should identify such type of instances as soon as possible and take immediate countermeasure to contain or eliminate attacks. The analysts can also check potential vulnerabilities associated with these instances and perform network hardening to prevent such attacks from happening again.

Instances with high dependency ranks but low infection probabilities. In this case, the chance of the object being infected is low. The high dependency ranks could be due to multiple dependents contributing to the extrinsic AssetRank value of the node. Even though the infection probability is low, the attack propagation can be severe as high AssetRank means multiple dependents on that node. If that node is attacked, it will be very easy for the attacker to propagate the attack throughout the system. For example, some instances have high ranks in Table 2, but are not included in Figure 11. This means these instances may have very high impact on other objects, but currently have low infection probabilities. One example is Instance 69.1, file */etc/nsswitch.conf*, which is used by applications to determine the sources from which to obtain name-service information in a range of categories, and in what order. Although it may have a low infection probability (0.01 as shown in Table 1), security analysts must examine this file carefully due to its impact (ranked No.4 in Table 2).

Instances with low dependency ranks but high infection probabilities. In this case, there is a high chance that the objects are infected despite the low dependency ranks. Analyzing these objects can help analysts understand which objects are impacted by attackers' activities and how the impact happens. It will also be helpful to check the dependency ranks of these objects to make sure such impact will not propagate further. To show the dependency ranks of the instances appearing on the zero-day attack path, we also provided the rank values of the top 50 nodes among these instances in Table 3. A large number of instances, such as Instance 2114.2 (file */home/user/testbed/workstation_attack/wunderbar_emporium/pwnkernel.c*), have a high infection probability but a relatively low dependency rank. The infection probability for Instance 2114.2 is 0.83, but its dependency rank is not even among the top 50 nodes. This type of nodes are not dependent

upon by many other nodes, but still need to be carefully checked due to their high infection probabilities.

Instances with low dependency ranks as well as low infection probabilities. In this case, there is a very small chance that the instance is infected. The low dependency rank implies that there are not many dependents of the object. Hence, if it is infected, the propagation will be in a controlled manner. The instances in this category can get the lowest priority when resources are limited.

Therefore, such analysis can help security analysts identify the most critical system objects in terms of dependency ranks and infection probabilities. Both perspectives give us a list of priority nodes which can be investigated first. This helps the analyst focus their limited time and scarce resources on the high priority objects.

6. Related Work

This paper is based on the prior works on zero-day attack path detection [1, 2, 8], which utilize the dependency relations among system objects to capture the attackers' activities at system level. [1] proposed to build system object dependency graph by analyzing the system calls, and then reveal the zero-day attack paths by tracking from system objects involved in security alerts. [2, 8] then proposed to build bayesian networks on top of the system object instance graph, compute the object instances' infection probabilities, and then connect the instances with high infection probabilities to reveal the zero-day attack paths. Both approaches focused on recognizing the system objects or instances that are potentially compromised, but did not investigate the objects or instances that are more impactful and may infect more instances.

This work is also related to system level provenance tracking, which also monitor system activities for the purpose of intrusion forensics or recovery. A number of works [16–26] tracks attackers' activities at the system level. Similar to those works, this paper also utilizes the dependencies among system objects, such as files, processes, and sockets. However, the main purpose of this paper is to recognize the impactful system instances based on their dependencies, rather than tracking the attackers' actions.

In terms of capturing dependencies and ranking dependency graphs, this work is also related to logical attack graphs [27–34]. Attack graphs are used to generate potential attack paths by analyzing the causalities relations involved in vulnerability exploitations. Dependency attack graphs is one type of attack graphs that capture dependencies among facts. For example, a fact that "vulnerability X is exploited" may de-

pend on the facts that "the host runs a service" and "the service has vulnerability X" etc. Based on the attack graphs, [1, 35] proposed to further rank the attack graphs to identify the critical attack assets in the enterprise networks, such as the important vulnerabilities on a server. Our paper also ranks the dependency graphs, but a different type of dependency graphs at a different abstract level. The works based on logical attack graphs are at the application level and identifies critical attack assets associated with applications or services, while our paper is at the system-level and identifies impactful system instances within a system.

7. Conclusion

System Object Instance Graph can reveal the attack paths by connecting object instances with high infection probabilities. However, the SOIG can be very overwhelming and difficult to comprehend. In the experiments used in this paper, the system calls were collected for a small timeframe. Despite the small timeframe, the number of system calls was huge. The complexity keeps increasing as more and more instances are created when objects are accessed by objects.

Combining the dependency ranks of instances with their infection probabilities can tremendously reduce the time and resources needed for understanding the otherwise complex graphs. This approach can be used in combination with many other intrusion detection tools to further narrow down the suspicious system objects and focus on the most critical nodes on the attack paths.

Acknowledgement

Xiaoyan Sun and Jun Dai are supported by NSF DGE-2105801.

References

[1] J. Dai, X. Sun, P. Liu, Patrol: Revealing zero-day attack paths through Network-Wide System Object Dependencies. ESORICS, Springer, Berlin, Heidelberg, 2013.

[2] X. Sun, J. Dai, P. Liu, A. Singhal, J. Yen, Towards Probabilistic Identification of Zero-day Attack Paths, IEEE Conference on Communications and Network Security (CNS), February 2017.

[3] R. Sawilla and X. Ou, Googling Attack Graphs, Defence Research and Development, Canada, September 2007.

[4] D. F. Gleich, PageRank Beyond the Web, Society for Industrial and Applied Mathematics Review, Society for Industrial and Applied

Mathematics, 57(3), 321-363 , August 2015.

[5] Essential Perl [Online]. Available: http://cslibrary.stanford.edu/108/EssentialPerl.html

[6] Graphviz User Guide [Online]. Available: https://graphviz.readthedocs.io/en/sTable/manual.html

[7] The DOT Language [Online]. Available: http://www.graphviz.org/doc/info/lang.html

[8] X. Sun, J. Dai, P. Liu, A. Singhal, and J. Yen. "Using Bayesian networks for probabilistic identification of zero-day attack paths." IEEE Transactions on Information Forensics and Security 13, no. 10 (2018): 2506-2521.

[9] L. Page. The PageRank Citation Ranking: Bringing Order to the Web, Technical Report. Stanford Digital Library Technologies Project, 1998.

[10] CVE-2008-0166. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166. Accessed Mar 1, 2018.

[11] CVE-2009-2692. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692. Accessed Mar 1, 2018.

[12] Snort. [Online]. Available: https://www.snort.org/. Accessed Mar 1, 2018.

[13] Tripwire. [Online]. Available: http://www.tripwire.com/. Accessed Mar 1, 2018.

[14] Wireshark. [Online]. Available: http://www.wireshark.org. Accessed Mar 1, 2018.

[15] Ntop. [Online]. Available: http://www.ntop.org. Accessed Mar 1, 2018.

[16] S. T. King, P. M. Chen. Backtracking intrusions. ACM SIGOPS 2003.

[17] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. SIGOPS Oper. Syst. Rev. 2005.

[18] A. Goel, W. C. Feng, D. Maier, W. C. Feng, J. Walpole. Forensix: a robust, high-performance reconstruction system. ICDCS 2005.

[19] K. M.-Reddy, D. A Holland, U. Braun, M I Seltzer. Provenance-aware storage systems. USENIX ATC 2006.

[20] U. Braun, S. Garfinkel, D. A Holland, K. M.-Reddy, M. I Seltzer. Issues in automatic provenance collection. International Provenance and Annotation Workshop 2006.

[21] A. Gehani, D.Tariq. Spade: support for provenance auditing in distributed environments. International Middleware Conference 2012.

[22] D. J Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-fi: collecting high-fidelity whole-system provenance. ACSAC 2012.

[23] A. Bates, D. J.Tian, K. RB Butler, T. Moyer. Trustworthy whole-system provenance for the linux kernel. USENIX Security 2015.

[24] X. Kiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, E. H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. ICDCS 2006.

[25] S. Ma, X. Zhang, D. Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. NDSS 2016.

[26] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. USENIX Security 2017.

[27] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. ACM workshop on visualization and data mining for computer security, 2004.

[28] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. Managing Cyber Threats. Springer, Boston, MA, 2005.

[29] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. ACM CCS 2002.

[30] K. Ingols, R. Lippmann, and K. Piwowarski. Practical attack graph generation for network defense. ACSAC 2006.

[31] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. ACSAC 2003.

[32] X. Ou, W. F. Boyer, and M. A. McQueen. A scalable approach to attack graph generation. ACM CCS 2006.

[33] X. Ou, S. Govindavajhala, A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. USENIX Security 2005.

[34] H.Huang, S. Zhang, X. Ou, A.Prakash, K. Sakallah. Distilling critical attack graph surface iteratively through minimum-cost sat solving. ACSAC 2011.

[35] R. E. Sawilla, and X. Ou. "Identifying critical attack assets in dependency attack graphs." *In European Symposium on Research in Computer Security*, pp. 18-34. Springer, Berlin, Heidelberg, 2008.