Towards Probabilistic Identification of Zero-day Attack Paths

Xiaoyan Sun[1], Jun Dai[2], Peng Liu[1], Anoop Singhal[3], John Yen[1]

[1] Penn State University, University Park, PA 16802, USA
[2] California State University, Sacramento, CA 95819, USA
[3] National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
xzs5052,pliu,jyen@ist.psu.edu, jun.dai@csus.edu, anoop.singhal@nist.gov

Abstract. Zero-day attacks continue to challenge the enterprise network security defense. A zero-day attack path is formed when a multi-step attack contains one or more zero-day exploits. Detecting zero-day attack paths in time could enable early disclosure of zero-day threats. In this paper, we propose a probabilistic approach to identify zero-day attack paths and implement a prototype system named Pr0bA. A *System Object Instance Dependency Graph* (*SOIDG*) is first built from system calls to capture the intrusion propagation. To further reveal the zero-day attack paths hiding in the SOIDG, our system constructs an SOIDG-based Bayesian network. By leveraging intrusion evidence, the Bayesian network can quantitatively compute the probabilities of object instances being infected. The object instances with high infection probabilities reveal themselves and form the candidate zero-day attack paths. The experiment results show that our system can successfully identify zero-day attack paths and the paths are of manageable size.

1 Introduction

Defending against zero-day attacks is one of the most fundamentally challenging problems yet to be solved. Zero-day attacks are usually enabled by unknown vulnerabilities. The information asymmetry between what the attacker knows and what the defender knows makes zero-day exploits extremely difficult to detect. Signature-based detection assumes that a signature is already extracted from detected exploits. Anomaly detection [1–3] may detect zero-day exploits, but this solution has to cope with high false positive rates.

Recently, one noticeable research progress is based on a key observation that in many cases identifying zero-day attack paths is substantially more feasible than identifying individual zero-day exploits. A *zero-day attack path* is a multi-step attack path which includes one or more zero-day exploits. When not every exploit in a zero-day attack path is zero-day, part of the path can already be detected by commodity signature-based IDS. That is, the defender can leverage one weakness of the attacker: in many cases he is unable to let an attack path be completely composed of zero-day exploits.

Both alert correlation [4, 5] and attack graphs [6–9] are limited in identifying zero-day attack paths. They both can identify the non-zero-day segments (i.e., "islands") of a zero-day attack path; however, none of them can automatically bridge these islands into a meaningful path, especially when different segments may belong to totally irrelevant attack paths.

To address these limitations, Dai et al. proposed to use (data and control) dependencies between OS-level objects (e.g., files, processes, sockets) to bridge the non-zero-day islands so that the zero-day segments can be revealed [10]. Nevertheless, this approach has a main limitation, namely the explosion in the number and size of zero-day attack path candidates. The forward and backward tracking from intrusion detection points can result in a large number of candidate paths, especially when lots of intrusion detection points are available. In addition, a candidate path can be too big because it preserves every tracking-reachable object. With the large number and size, discerning from the candidates and verifying the real zero-day attack paths becomes unpractical. As a consequence, in many cases this approach may generate a big "haystack" for the defender to find a "needle" in it.

In this paper, we propose a probabilistic zero-day attack path identification approach to address the explosion problem. The goal is to make the "haystack" orders of magnitude smaller. Our approach is to 1) establish a *System Object Instance Dependency Graph* (*SOIDG*) to capture the intrusion propagation, where an instance of an object is a "version" of the object with a specific timestamp; 2) build a Bayesian network (BN) based on the SOIDG to leverage the intrusion evidence collected from various information sources. Intrusion evidence can be the abnormal system and network activities that are noticed by human admins or security sensors such as Intrusion Detection Systems (IDSs). With the evidence, the SOIDG-based BN can quantitatively compute the probabilities of object instances being infected. Connected through dependency relations, the instances with high infection probabilities form a path, which can be viewed as a candidate zero-day attack path. As a result, the SOIDG-based BN can significantly narrow down the set of suspicious objects and make the manual verification of the zero-day attack paths feasible.

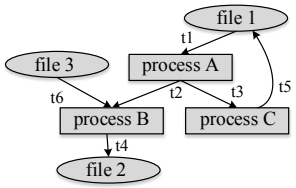
This approach is proposed based on the following insights: 1) A BN is able to capture cause-and-effect relations, and thus can be used to model the infection propagation among instances of different system objects: the cause is an already infected instance of one object, while the effect is its infection to an innocent instance of another object. We name this cause-and-effect relation as a type of *infection causality*, which is formed due to the interaction between the two objects in a system call operation. 2) An SOIDG can reflect the infection propagation process by capturing the dependencies among instances of different system objects. 3) Based on above insights, a BN can be constructed on top of the SOIDG because they couple well with each other: the dependencies among instances of different system objects can be directly interpreted into infection causalities in the BN. The BN's graphical nature makes it fit well with SOIDG.

We made the following contributions.

- To the best of our knowledge, this work is the first probabilistic approach towards zero-day attack path identification.
- We proposed constructing Bayesian network at the low system object level by introducing System Object Instance Dependency Graph.
- We have designed and implemented a system prototype named Pr0bA, which can successfully identify the zero-day attack paths.

```
t1: process A reads file 1
t2: process A creates process B
t3: process A creates process C
t4: process B writes file 2
t5: process C writes file 1
t6: process B reads file 3
```

(a) simplified system call log in time-order



(b) SODG

Fig. 1: An SODG generated by parsing an example set of simplified system call log. The label on each edge shows the time associated with the corresponding system call.

2 Motivation and Approach Overview

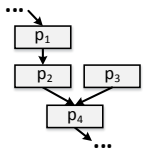
2.1 System Object Dependency Graph

This paper classifies OS-level entities in UNIX-like systems into three types of objects: processes, files and sockets. The operating system performs a set of operations towards these objects via system calls such as read, write, etc. For instance, a process can read from a file as input, and then write to a socket. Such interactions among system objects enable intrusions to propagate from one object to another. Generally an intrusion starts with one or several seed objects that are created directly or indirectly by attackers. The intrusion seeds can be processes such as compromised service programs, or files such as viruses, or corrupted data, etc. As the intrusion seeds interact with other system objects via system call operations, the innocent objects can get infected. We call this process as *infection propagation*. Therefore the intrusion will propagate throughout the system, or even propagate to the network through socket communications.

To capture the intrusion propagation, previous work [10, 16, 17] has explored constructing System Object Dependency Graphs (SODGs) by parsing system call traces. Each system call is interpreted into three parts: a source object, a sink object, and a dependency relation between them. The objects and the dependencies respectively become nodes and directed edges in SODGs. For example, a process reading a file in the system call *read* indicates that the process (sink) depends on the file (source). The dependency is denoted as *file→process*. Similar rules (Table 5 in Appendix) as used in previous work [10, 16, 17] can be adopted to generate dependencies from system calls. Fig. 1b is an example SODG generated by parsing the simplified system call log shown in Fig. 1a.

2.2 Why use Bayesian Network?

The SODG can be used directly to identify the candidate zero-day attack paths through forward and backward tracking from intrusion detection points. However, such tracking will result in an explosion in the number and size of zero-day attack path candidates. The explosion is two-fold. First, in addition to real zero-day attack paths, the number of false positive path candidates is proportional to the number of false alerts. Second, an individual candidate path may contain too many objects for security analysts to comprehend, because it preserves every tracking-reachable object. Therefore, discerning from the candidates and verifying the real paths becomes difficult.



CPT at node p_2

	p_1=T	p_1=F
p_2=T	0.9	0.01
p_2=F	0.1	0.99

Fig. 2: An example Bayesian network.

A Bayesian network can effectively deal with the explosion problem. The key reason is that a BN can quantitatively compute the probabilities of objects being infected through incorporating intrusion evidence from a variety of information sources. By only focusing on the objects with high infection probabilities, the set of suspicious objects can be significantly narrowed down. The candidate zero-day attack paths formed by the high-probability objects through dependency relations can be of manageable size.

The BN is a probabilistic graphical model that represents the cause-and-effect relations. It is formally defined as a Directed Acyclic Graph (DAG) that contains a set of nodes and directed edges, where a node denotes a variable of interest, and an edge denotes the causality relations between two nodes. The strength of such causality relation is indicated using a conditional probability table (CPT). Fig. 2 shows an example BN and the CPT tables associated with p_2. Given p_1 is true, the probability of p_2 being true is 0.9, which can be represented with $P(p_2 = T|p_1 = T) = 0.9$. Similarly, the probability of p_4 can be determined by the states of p_2 and p_3 according to a CPT table at p_4. BN is able to incorporate the collected evidence by updating the posterior probabilities of interested variables. For example, after evidence $p_2 = T$ is observed, it can be incorporated by computing probability $P(p_1 = T|p_2 = T)$.

2.3 Problems of Constructing Bayesian Network based on SODG

SODG has the potential to serve as the base of BN construction. For one thing, BN has the capability of capturing cause-and-effect relations in infection propagation. For another thing, SODG reflects the dependency relations among system objects. Such dependencies imply and can be leveraged to construct the infection causalities in BN. For example, the dependency *process A→file 1* in an SODG can be interpreted into an infection causality relation in BN: *file 1* is likely to be infected if *process A* is already infected. In such a way, an SODG-based BN can be constructed by directly taking the structure topology of SODG.

However, several drawbacks of the SODG prevent it from being the base of BN. First, an SODG without time labels cannot reflect the correct information flow according to the time order of system call operations. This is a problem because the time labels cannot be preserved when constructing BNs based on SODGs. Lack of time information will cause incorrect causality inference in the SODG-based BNs. For example, without the time labels, the dependencies in Fig. 1b indicates infection causality relations existing among *file 3*, *process B* and *file 2*, meaning that if *file 3* is infected, *process B* and *file 2* are likely to be infected by *file 3*. Nevertheless, the time information shows that the system call operation *"process B reads file 3"* happens at time *t6*, which is after the

operation *"process B writes file 2"* at time *t4*. This implies that the status of *file 3* has no direct influence on the status of *file 2*.

Second, the SODG contains cycles among nodes. For instance, *file 1*, *process A* and *process C* in Fig. 1b form a cycle. By directly adopting the topology of SODG, the SODG-based BN inevitably inherits cycles from SODG. However, the BN is an *acyclic* probabilistic graphical model that does not allow any cycles.

Third, a node in an SODG can end up with having too many parent nodes, which will render the CPT assignment difficult and even impractical in the SODG-based BN. For example, if *process B* in Fig. 1b continuously reads hundreds of files (which is normal in a practical operating system), it will get hundreds of file nodes as its parents. In the corresponding SODG-based BN, if each file node has two possible states that are *"infected"* and *"uninfected"*, and the total number of parent file nodes are denoted as n, then the CPT table at *process B* has to assign 2^n numbers in order to specify the infection causality of the parent file nodes to *process B*. This is impractical when n is very large.

To address the above problems, we propose a new type of dependency graph, *System Object Instance Dependency Graph*, which is a mutation of SODG.

2.4 System Object Instance Dependency Graph

In SOIDG, each node is not an object, but an instance of the object with a certain timestamp. Different instances are different "versions" of the same object at different time points, and can thus have different infection status.

Definition 1. *System Object Instance Dependency Graph (SOIDG)*
If the system call trace in a time window $T[t_{begin}, t_{end}]$ is denoted as Σ_T and the set of system objects (mainly processes, files or sockets) involved in Σ_T is denoted as O_T, then the SOIDG is a directed graph $G_T(V, E)$, where:

- V is the set of nodes, and initialized to empty set $\varnothing$;
- E is the set of directed edges, and initialized to empty set $\varnothing$;
- If a system call $syscall \in \Sigma_T$ is parsed into two system object instances src_i, $sink_j$, $i, j \geq 1$, and a dependency relation dep_c: $src_i \rightarrow sink_j$ (according to dependency rules in Table 5), where src_i is the i^{th} instance of system object $src \in O_T$, and $sink_j$ is the j^{th} instance of system object $sink \in O_T$, then $V = V \cup \{src_i, sink_j\}$, $E = E \cup \{dep_c\}$. The timestamps for *syscall*, dep_c, src_i, and $sink_j$ are respectively denoted as *t_syscall, t_*dep_c*,t_*src_i*, and t_*$sink_j$. The *t_*dep_c inherits *t_syscall* from *syscall*. The indexes i and j are determined before adding src_i and $sink_j$ into V by:
  - For $\forall\ src_m, sink_n \in V$, $m, n \geq 1$, if i_{max} and j_{max} are respectively the maximum indexes of instances for object *src* and *sink*, and;
  - If $\exists\ src_k \in V$, $k \geq 1$, then $i = i_{max}$, and *t_*src_i stays the same; Otherwise, $i = 1$, and *t_*src_i is updated to *t_syscall*;
  - If $\exists\ sink_z \in V$, $z \geq 1$, then $j = j_{max}+1$; Otherwise, $j = 1$. In both cases *t_*$sink_j$ is updated to *t_syscall*; If $j \geq 2$, then $E = E \cup \{dep_s: sink_{j-1} \rightarrow sink_j\}$.
- If $a \rightarrow b \in E$ and $b \rightarrow c \in E$, then c transitively depends on a.

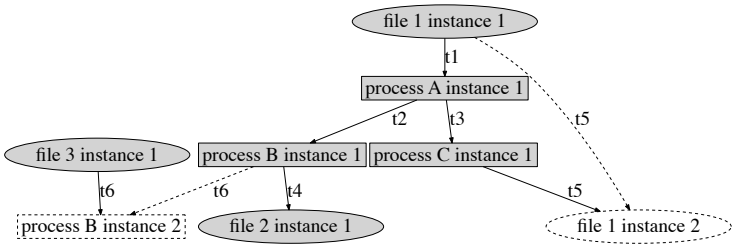


Fig. 3: An SOIDG generated by parsing the same set of simplified system call log as in Fig. 1a. The label on each edge shows the time associated with the corresponding system call operation. The dotted rectangle and ellipse are new instances of already existed objects. The solid edges and the dotted edges respectively denote the contact dependencies and the state transition dependencies.

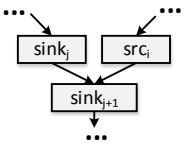
According to Definition 1, for *src* object, a new instance is created only when no instances of *src* exist in the SOIDG. For *sink* object, however, a new instance is created whenever a *src*→*sink* dependency appears. The underlying insight is that the status of the *src* object will not be altered by the *src*→*sink*, while the status of *sink* will be influenced. Hence a new instance for an object should be created when the object has the possibility of being affected. A dependency dep_c is added between the most recent instance of *src* and the newly created instance of *sink*. We name dep_c as *contact dependency* because it is generated by the contact between two different objects through a system call operation.

In addition, when a new instance is created for an object, a new dependency relation dep_s is also added between the most recent instance of the object and the new instance. This is necessary and reasonable because the status of the new instance can be influenced by the status of the most recent instance. We name dep_s as *state transition dependency* because it is caused by the state transition between different instances of the same system object.

The SOIDG can well tackle the problems existing in the SODG for constructing BNs. It can be illustrated using Fig. 3, an SOIDG created for the same simplified system call log as in Fig. 1a. First, the SOIDG is able to reflect correct information flows by implying time information through creating object instances. For example, instead of parsing the system call at time *t6* directly into *file 3*→*process B*, Fig. 3 parsed it into *file 3 instance 1*→*process B instance 2*. Comparing to Fig. 1b in which *file 3* has indirect infection causality on *file 2* through *process B*, the SOIDG in Fig. 3 indicates that *file 3* can only infect *instance 2* of *process B* but no previous instances. Hence in this graph *file 3* does not have infection causality on *file 2*.

Second, SOIDGs can break the cycles contained in SODGs. Again, in Fig. 3, the system call at time *t5* is parsed into *process C instance 1*→*file 1 instance 2*, rather than *process C*→*file 1* as in Fig. 1b. Therefore, instead of pointing back to *file 1*, the edge from process C is directed to a new instance of *file 1*. As a result, the cycle formed by *file 1, process A* and *process C* is broken.

Third, the mechanism of creating new *sink* instances for a relation *src*→*sink* prevents the nodes in SOIDGs from getting too many parents. For example,



CPT at node $sink_{j+1}$

	$sink_j$=Infected		$sink_j$=Uninfected	
	src_i=Infected	src_i=Uninfected	src_i=Infected	src_i=Uninfected
$sink_{j+1}$=Infected	1	1	τ	ρ
$sink_{j+1}$=Uninfected	0	0	$1-\tau$	$1-\rho$

Fig. 4: The infection propagation models.

process B instance 2 in Fig. 3 has two parents: *process B instance 1* and *file 3 instance 1*. If *process B* appears again as the *sink* object in later *src→sink* dependencies, new instances of *process B* will be created instead of directly adding *src* as the parent to *process B instance 2*. Therefore, a node in the SOIDG only has 2 parents at most: one is the previous instance for the same object; the other one is an instance for a different object that the node depends on.

3 SOIDG-based Bayesian Networks

To build a BN based on an SOIDG and compute probabilities for interested variables, two steps are required. First, the CPT tables have to be specified for each node via constructing proper infection propagation models. Second, evidence from different information sources has to be incorporated into BN for subsequent probability inference.

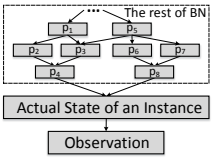
3.1 The Infection Propagation Models

In SOIDG-based BNs, each object instance has two possible states, *"infected"* and *"uninfected"*. The strength of the infection causalities among the instances has to be specified in corresponding CPT tables. Our infection propagation models in this paper deal with two types of infection causalities, *contact infection causalities* and *state transition infection causalities*, which correspond to the contact dependencies and state transition dependencies in SOIDGs.

Contact Infection Causality Model. This model captures the infection propagation between instances of two different objects. Fig. 4 shows a portion of BN constructed when a dependency *src→sink* occurs and the CPT table associated with $sink_{j+1}$. When $sink_j$ is uninfected, the probability of $sink_{j+1}$ being infected depends on the infection status of src_i, a *contact infection rate* τ and an *intrinsic infection rate* ρ, $0 \leq \tau, \rho \leq 1$.

The intrinsic infection rate ρ decides how likely $sink_{j+1}$ gets infected given src_i is uninfected. In this case, since src_i is not the infection source of $sink_{j+1}$, if $sink_{j+1}$ is infected, it should be caused by other factors. So ρ can be determined by the prior probabilities of an object being infected, which is usually a very small constant number.

The contact infection rate τ determines how likely $sink_{j+1}$ gets infected when src_i is infected. The value of τ determines to which extent the infection can be propagated within the range of an SOIDG. In an extreme case where $\tau = 1$, all the object instances will get contaminated as long as they have contact with the infected objects. In another extreme case where $\tau = 0$, the infection will be confined inside the infected object and does not propagate to any other contacting object instances. Our system allows security experts to tune the value of τ based on their knowledge and experience.



CPT at node Observation

	Actual=Infected	Actual=Uninfected
Observation=True	0.9	0.15
Observation=False	0.1	0.85

False negative rate

False positive rate

Fig. 5: Local observation model.

Since a large number of system call traces with ground truths are often unavailable, it is very unlikely to learn the parameters of τ and ρ using statistical techniques. Hence, currently these parameters have to be assigned by security experts. We will evaluate the impact of τ and ρ in Section 6.2.

State Transition Infection Causality Model. This model captures the infection propagation between instances of the same objects. We follow one rule to model this type of causalities: an object will never return to the state of *"uninfected"* from the state of *"infected"*[4]. That is, once an instance of an object gets infected, all future instances of this object will remain the infected state, regardless of the infection status of other contacting object instances. This rule is enforced in the CPT tables as exemplified in Fig. 4. If $sink_j$ is infected, the infection probability of $sink_{j+1}$ keeps to be 1, no matter whether src_i is infected or not. If $sink_j$ is uninfected, the infection probability of $sink_{j+1}$ is decided by the infection status of src_i according to the contact infection causality model.

3.2 Evidence Incorporation

BN is able to incorporate security alerts from a variety of information sources as the evidence of attack occurrence. Numerous ways have been developed to capture intrusion symptoms, which can be caused by attacks exploiting both known vulnerabilities and zero-day vulnerabilities. A tool Wireshark [12] can notice a back telnet connection that is instructed to open; an IDS such as Snort [13] may recognize a malicious packet; a packet analyzer tcpdump [14] can capture suspicious network traffic, etc. In addition, human security admins can also manually check the system or network logs to discover other abnormal activities that cannot be captured by security sensors. As more evidence is fed into BN, the identified zero-day attack paths get closer to real facts.

In this paper, we adopt two ways to incorporate evidence. First, add evidence directly on a node by providing the infection state of the instance. If human security experts have scrutinized an object and proven that an object is infected at a specific time, they can feed the evidence to the SOIDG-based BN by directly changing the infection status of the corresponding instance into *infected*. Second, leverage the local observation model (LOM) [22] to model the uncertainty towards observations. Human security admins or security sensors may notice suspicious activities that imply attack occurrence. Nonetheless, these observations often suffer from false rates. As shown in Fig. 5, an observation node can be added as the direct child node to an object instance. The implicit causality

[4] This rule is formulated based on the assumptions that no intrusion recovery operations are performed and attackers only conduct malicious activities.

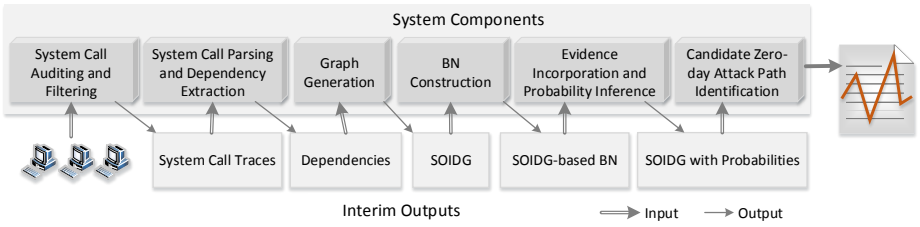


Fig. 6: System design.

relation is that the actual state of the instance can likely affect the observation to be made. If the observation comes from security alerts, the CPT inherently indicates the false rates of the security sensors. For example, $P(Observation = True \mid Actual = Uninfected)$ shows the false positive rate and $P(Observation = False \mid Actual = Infected)$ indicates the false negative rate.

4 System Design

Fig. 6 shows the overall system design, which includes 7 components.

System call auditing and filtering. System call auditing is performed against all running processes and should preserve sufficient OS-aware information. Subsequent system call reconstruction can thus accurately identify the processes and files by their process IDs or file descriptors. The filtering process basically prunes system calls that involve redundant and very likely innocent objects, such as the dynamic linked library files or some dummy objects. We conduct system call auditing at run time towards each host in the enterprise network.

System call parsing and dependency extraction. The collected system call traces are then sent to a central machine for off-line analysis, where the dependency relations between system objects are extracted according to Table 5.

Graph generation. The extracted dependencies are then analyzed line by line for graph generation. The generated graph can be either host-wide or network-wide, depending on the analysis scope. A network-wide SOIDG can be constructed by concatenating individual host-wide SOIDGs through instances of the communicating sockets. Algorithm 1 is the basis algorithm for SOIDG generation, which is designed according to the logic in Definition 1.

BN construction. The BN is constructed by taking the topology of an SOIDG. The instances and dependencies in an SOIDG become nodes and edges in BN. Basically the nodes and the associated CPT tables are specified in a *.net* file, which is one file type that can carry the SOIDG-based BN.

Evidence incorporation and probability inference. Evidence is incorporated by either providing the infection state of the object instance directly, or constructing an local observation model (LOM) for the instance. After probability inference, each node in the SOIDG receives a probability.

Candidate Zero-day Attack Paths Identification. To reveal the candidate zero-day attack paths from the mess of SOIDG, the nodes with high probabilities are to be preserved, while the link between them should not be broken. We implemented Algorithm 2 on the basis of depth-first search (DFS) algorithm [24]

to tag each node in the SOIDG as either possessing high probability itself, or having both an ancestor and a descendant with high probabilities. The tagged nodes are the ones that actually propagate the infection through the network, and thus should be preserved in the final graph. Our system allows a probability threshold to be tuned for recognizing high-probability nodes. For example, if the threshold is set at 80%, only instances that have the infection probabilities of 80% or higher will be recognized as the high-probability nodes.

5 Implementation

The whole system includes online system call auditing and off-line data analysis. For system call auditing, we share with Patrol [10] the same component that is implemented with a loadable kernel module. For the off-line data analysis, our prototype is implemented with approximately 2915 lines of gawk code that constructs a *.net* file for the SOIDG-based BN and a *dot*-compatible file for visualizing the candidate zero-day attack paths in Graphviz [25], and 145 lines of Java code for probability inference, leveraging the API provided by the BN tool SamIam [23].

An SOIDG can be too large due to the introduction of instances. Therefore, in addition to system call filtering, we also develop several ways to prune that SOIDGs while not impede reflecting the major infection propagation process.

One helpful way is to ignore the repeated dependencies. It is common that the same dependency may happen between two system objects for a number of times, even through different system call operations. For example, *process A* may write *file 1* for several times. In such cases, each time the *write* operation occurs, a new instance of *file 1* is created and a new dependency is added between the most recent instance of *process A* and the new instance of *file 1*. If the status of *process A* is not affected by any other system objects during this time period, the infection status of *file 1* will not change neither. Hence the new instances of *file 1* and the related new dependencies become redundant information in understanding the infection propagation. Therefore, a repeated $src \rightarrow sink$ dependency can be ignored if the *src* object is not influenced by other objects since the last time that the same $src \rightarrow sink$ dependency appeared.

Another way to simplify an SOIDG is to ignore the root instances whose original objects have never appear as the *sink* object in a $src \rightarrow sink$ dependency during the time period of being analyzed. For instance, *file 3* in Fig. 3 only appears as the *src* object in the dependencies parsed from the system call log in Fig. 1a, so *file 3 instance 1* can be ignored in the simplified SOIDG. Such instances are not influenced by other objects in the specified time window, and thus are not manipulated by attackers, neither. Hence ignoring these root instances does not break any routes of intrusion sequence and will not hinder the understanding of infection propagation. This method is helpful for situations such as a process reading a large number of configuration or header files.

A third way to prune an SOIDG is to ignore some repeated mutual dependencies, in which two objects will keep affecting each other through creating new instances. One situation is that a process can frequently send and receive

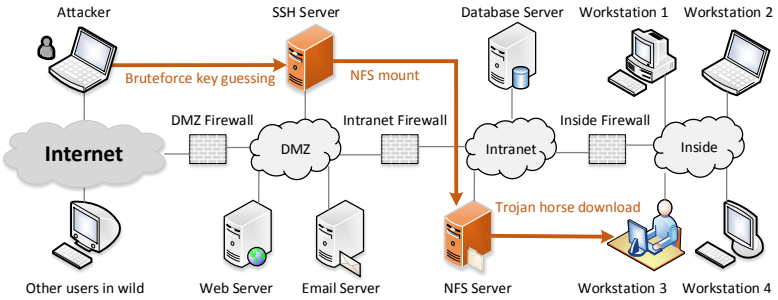


Fig. 7: Attack scenario.

messages from a socket. For example, in one of our experiments, 107 new instances are created respectively for the process (*pid:6706, pcmd:sshd*) and the socket (*ip:192.168.101.5, port: 22*) due to their interaction. Since no other objects are involved during this procedure, the infection status of these two objects will keep the same through all the new instances. Thus a simplified SOIDG can preserve the very first and last dependencies while neglect the middle ones. Another situation is that a process can frequently take input from a file and then write the output to it again after some operations. The middle repeated mutual dependencies could also be ignored in a similar way.

6 Experiments

6.1 Attack Scenario

We built a test-bed network and launched a three-step attack towards it. Fig. 7 illustrates the attack scenario, which is similar to the one in [10]. Step 1, the attacker exploits vulnerability CVE-2008-0166 to gain root privilege on SSH Server through a brute-force key guessing attack. Step 2, since the export table on NFS Server is not set up appropriately, the attacker can upload a malicious executable file to a public directory on NFS. The malicious file contains a Trojan-horse that can exploit CVE-2009-2692. The public directory is shared among all the hosts in the test-bed network. Step 3, once the malicious file is mounted and installed on the Workstation 3, the attacker is able to execute arbitrary code on Workstation 3. To capture the intrusion evidence for subsequent BN probability inference, we deployed security sensors in the test-bed, such as firewalls, Snort, Tripwire, Wireshark, Ntop [26] and Nessus. For sensors that need configuration, we tailored their rules or policy files to match our hosts.

Since zero-day exploits are not readily available, we emulate zero-day vulnerabilities with known vulnerabilities. For example, we treat CVE-2009-2692 as a zero-day vulnerability by assuming the current time is Dec 31, 2008. In addition, the configuration error on NFS is also viewed as a special type of unknown vulnerability because it is ruled out by vulnerability scanners like Nessus. The strategy of emulation also brings another benefit. The information for these "*known* zero-day" vulnerabilities can be available to verify the correctness of our experiment results.

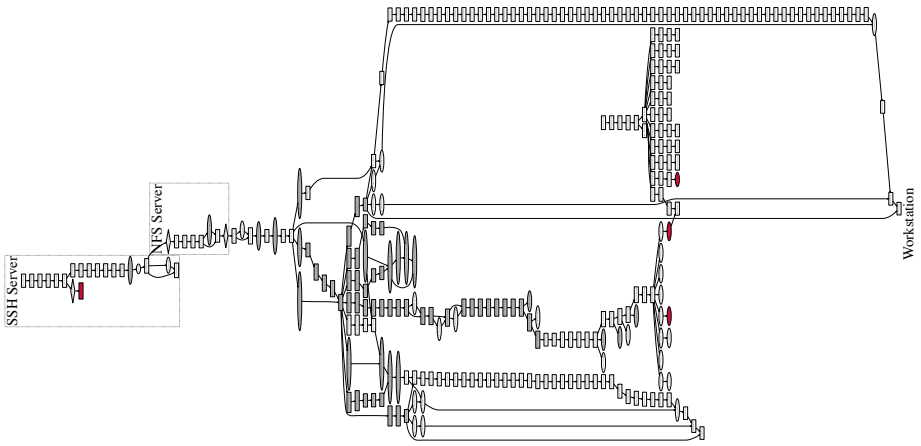


Fig. 8: The zero-day attack path in the form of SOIDG.

6.2 Experiment Results

While conducting the three-step attack, we simultaneously log the system calls on each host and collect the security alerts. After analyzing a total number of 143120 system calls generated by three hosts, an SOIDG-based BN with 1853 nodes and 2249 edges is constructed. The evidence as in Table 1 is collected and fed into BN.

Correctness. Given the evidence, Fig. 8 illustrates the identified candidate zero-day attack paths in the form of an SOIDG, with the contact infection rate τ as 0.9, the intrinsic infection rate ρ as 0.001, and the probability threshold of recognizing high-probability nodes as 80%. The processes, files, and sockets are denoted with rectangles, ellipses, and diamonds respectively. We mark the evidence with red color and the nodes that are verified to be malicious with grey color. Therefore, Fig. 8 shows that our approach can successfully reveal the actual zero-day attack path. It is worth noting that although no evidence is provided on NFS Server, but the identified attack path can still demonstrate how NFS Server contributes to the overall intrusion propagation: the file *workstation_attack.tar.gz* is uploaded from SSH Server to the */exports* directory on NFS Server, and then downloaded to */mnt* on Workstation 3. More importantly, the identified path can expose key objects that are related to the exploits of zero-day vulnerabilities. For example, the identified system objects on NFS Server can alert system admins for possible configuration errors because SSH Server should not have the privilege of writing to the */exports* directory. As another example, the object *PAGE0: memory(0-4096)* on Workstation is also exposed as highly suspicious on the identified attack path. Page-zero is actually what triggers the null pointer dereference and enables attackers gain privilege on Workstation 3. Therefore, exposing the page-zero object can help system admins to further diagnose how the intrusion happens and propagates.

An additional merit of our approach is that the SOIDG-based BN can clearly show the state transitions of an object using instances. By matching the in-

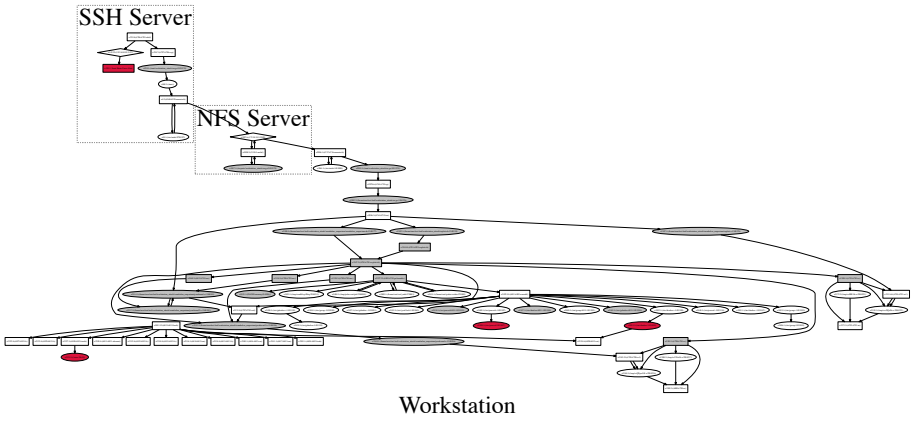


Fig. 9: The zero-day attack path in the form of SODG.

stances and dependencies back to the system call traces, it can even find out the exact system call that causes the state-changing of the object. For example, the node $x2086.4$*:(6763:6719:tar)* in Fig. 8 represents the fourth instance of process (*pid:6763, pcmd:tar*). Previous instances of the process are considered as innocent because of their low infection probabilities. The process becomes highly suspicious only after a dependency occurs between node $x2082.2$*:(/home/user/test-bed/workstation_attack.tar.gz:1384576)* and node $x2086.4$. Matching the dependency back to the system call traces reveals that the state change of the process is caused by "*syscall:read, start:827189, end:827230, pid:6763, ppid:6719, pcmd:tar, ftype:REG, pathname:/home/user/test-bed/workstation_attack.tar.gz, inode:1384576*", a system call indicating that the process reads a suspicious file.

Table 1: The Collected Evidence

ID	Host	Evidence
E1	SSH Server	Snort messages "potential SSH brute force attack"
E2	Workstation	Tripwire reports "/virus is added"
E3	Workstation	Tripwire reports "/etc/passwd is modified"
E4	Workstation	Tripwire reports "/etc/shadow is modified"

Size of Candidate Zero-day Attack Paths. If all the instances belonging to the same object are merged into one node, we will generate a zero-day attack path in the form of SODG as shown in Fig. 9. This path contains only objects and can be used for verification when details regarding instances are not needed. The main candidate path identified by Patrol contains 175 objects, while the path by our system is composed of only 77 objects, and thus can be verified with ease. Considering that the total number of objects involved in original SOIDG is only 913, the 56% reduction of path size is substantial. Our further investigation shows that when the time period of being analyzed is longer, our system can generate candidate paths much smaller than Patrol without hurting the correctness of the paths.

Influence of Evidence. We choose a number of nodes in Fig. 8 as the representative interested instances. Table 2 shows how the infection probabilities

of these instances change after each piece of evidence is fed into BN. We assume the evidence is observed in the order of attack sequence. The results show that when no evidence is available, the infection probabilities for all nodes are very low. When *E1* is added, only a few instances on SSH Server receive probabilities higher than 60%. After *E2* is observed, the infection probabilities for instances on Workstation 3 increase, but still not much. As *E3* and *E4* arrive, 5 of the 9 representative instances on all three hosts become highly suspicious. Therefore, the evidence makes the instances on the actual attack paths emerge gradually from the "sea" of instances in the SOIDG. However, it is also possible that the arrival of some evidence may decrease the probabilities of certain instances, so that these instances will get removed from the final path. In a word, as more evidence is collected, the revealed zero-day attack paths become closer to the actual fact.

Table 2: The Influence of Evidence

Evidence	***SSH Server***			***NFS Server***		***Workstation***			
	x4.1	x10.1	x253.3	x1007.1	x1017.1	x2006.2	x2083.1	x2108.1	x2311.32
No Evi.	0.56%	0.51%	0.57%	0.51%	0.54%	0.54%	0.51%	0.51%	1.21%
E1	63.76%	57.38%	79.13%	57.38%	46.54%	41.92%	37.75%	24.89%	26.93%
E2	63.76%	57.38%	79.13%	57.38%	46.94%	42.58%	38.34%	27.04%	30.09%
E3	86.82%	78.14%	80.76%	84.50%	75.63%	81.26%	79.56%	75.56%	81.55%
E4	86.84%	78.16%	80.77%	84.53%	75.65%	81.3%	79.59%	75.60%	81.66%

Influence of False Alerts. We assume that *E4* is a false alarm generated by Tripwire and evaluate its influence to the BN output. Table 3 shows that when only one piece of evidence exists, the observation of *E4* will at least greatly influence the probabilities of some instances on Workstation 3. However, when other evidence is fed into BN, the influence of *E4* decreases. For instance, given just *E1*, the infection probability of $x2006.2$ is 97.78% when *E4* is true, but should be 29.96% if *E4* is a false alert. Nonetheless, if all other evidence is already input into BN, the infection probability of $x2006.2$ only changes from 81.13% to 81.3% if *E4* becomes a false alert. Therefore, the impact of false alerts can be reduced substantially if sufficient evidence is collected.

Table 3: The Influence of False Alerts

Evidence		x4.1	x10.1	x253.3	x1007.1	x1017.1	x2006.2	x2083.1	x2108.1	x2311.32
Only E1	E4=True	98.46%	88.62%	81.59%	98.20%	88.30%	97.78%	97.67%	90.23%	94.44%
	E4=False	56.33%	50.70%	78.60%	48.65%	37.60%	29.96%	24.92%	10.89%	12.48%
All Evidence	E4=True	86.84%	78.16%	80.77%	84.53%	75.65%	81.3%	79.59%	75.60%	81.66%
	E4=False	86.74%	78.06%	80.76%	84.41%	75.54%	81.13%	79.42%	75.39%	81.38%

Sensitivity Analysis and Influence of τ and ρ. We also performed sensitivity analysis and evaluated the impact of the contact infection rate τ and the intrinsic infection rate ρ by tuning these numbers. ρ is usually set at a very low value, so our experiment results are not very sensitive to the value of ρ. Since τ decides how likely $sink_j$ get infected given src_i is infected in a $src_i \rightarrow sink_j$ dependency, the value of τ will definitely influence the probabilities produced by BN. If a node is marked as infected, other nodes that are directly or indirectly connected to this node should expect higher infection probabilities when

τ is bigger. Our experiments show that adjusting τ within a small range (e.g. changing from 0.9 to 0.8) does not influence the output probabilities much, but a major adjustment of τ (e.g. changing it from 0.9 to 0.5) can largely affect the probabilities. However, we still argue that although τ influences the produced infection probabilities, it will not greatly affect the identification of zero-day attack paths. Our rationale is that the probability threshold of recognizing high-probability nodes for zero-day attack paths can be adjusted according to the value of τ. For example, when τ is a small number such as *50%*, even nodes that have low infection probabilities of around *40%* to *60%* should be considered as highly suspicious because it is hard for an instance to get infected with such a low contact infection rate.

Complexity. One concern of adopting SOIDG is that it can become too large due to introduction of instances. However, the techniques of pruning the SOIDG can significantly reduce the number of instances. Table 4 summarizes the total number of instances in SOIDGs for each host before and after the pruning. It shows that the number of instances can be reduced to an acceptable value.

The experiment results also show that the off-line data analysis is very efficient. Considering that our system shares the system call logging component with Patrol, we will not repeat the evaluation of its run-time performance overhead. We only evaluate time cost for the off-line data analysis, which includes the time for SOIDG-based BN generation, probability inference and zero-day attack path identification. The time cost for probability inference depends on the algorithm employed in SamIam. The time complexity can be $O(|V|^2)$ for both SOIDG-based BN generation and zero-day attack path identification, because the DFS algorithm is applied towards every node in the SOIDG. For our experiments, Table 4 already shows the time required for constructing the SOIDG-based BN for each host, so the total time of BN construction comes to around 27 seconds. For a BN with approximately 1854 nodes, assuming that the evidence is already fed into BN and the algorithm used is *recursive conditioning*, the average time cost is 1.57 seconds for BN compilation and probability inference, and 59 seconds for zero-day attack path identification. Combining all the time required together, the average data analysis speed is 280 KB/s, which is reasonable comparing to the system call generation speed of around 1.03 KB/s [10]. The average memory used for compiling a BN with approximately 1854 nodes is 4.32 Mb.

Table 4: The Impact of Pruning the SOIDG

	SSH Server		***NFS Server***		***Workstation***	
	before	after	before	after	before	after
number of syscalls in raw data trace	82133		14944		46043	
size of raw data trace (MB)	13.8		2.3		7.9	
number of extracted object dependencies	10310		11535		17516	
number of objects	349		20		544	
number of instances(nodes) in SOIDG	10447	745	11544	39	17849	1069
number of dependencies(edges) in SOIDG	20186	968	19863	37	34549	1244
number of contact dependencies	9888	372	8329	8	17033	508
number of state transition dependencies	10298	596	11534	29	17516	736
average time for graph generation(s)	14	11	6	5	13	11
.net file size(KB)	2000	123	2200	8	3600	180

7 Related Work

The work that is most related to us is Patrol. Our work differs from Patrol in several aspects. First, Patrol relies on "shadow indicators" to distinguish zero-day attack paths from other candidate paths. However, investigating and crafting shadow indicators requires human analysts' or even the whole community's efforts. Instead, our approach solely relies on the collected intrusion evidence to generate a zero-day attack path. Second, Patrol identifies the candidate zero-day attack paths by tracking from a trigger point. If the trigger points are provided by security sensors with high false rates, the identified paths can also suffer from certain false rates. In constrast, our system does not perform any tracking, but only relies on the computed probabilities. By taking various evidence in, the SOIDG-based BN can cope with false rates to a large extent. Third, Patrol only conducts qualitative analysis and treats every object on the identified paths as having the same malicious status. Scrutinizing every object on the path to verify its status is a daunting job, especially when the identified path is very big. Compared to Patrol, the SOIDG-based BN quantifies the infection status of system objects with probabilities. By only focusing on system objects with relatively high probabilities, we can significantly reduce the set of suspicious objects, and make the subsequent verification of zero-day attack paths practical.

Other related work includes system call dependency tracking and zero-day attack identification. System call dependency tracking is first proposed in [16] to help the understanding of intrusion sequence. It is then applied for alert correlation in [4, 5]. Instead of directly correlating these alerts, our system takes the alerts as evidence and quantitatively compute the infection probabilities of system objects. [27] conducts an empirical study to reveal the zero-day attacks by identifying the executable files that are linked to exploits of known vulnerabilities. A zero-day attack is identified if a malicious executable is found before the corresponding vulnerability is disclosed. Attack graphs have been employed to measure the security risks caused by zero-day attacks [19–21]. Nevertheless, the metric simply counts the number of required unknown vulnerabilities for compromising an asset, rather than detects the actually occurred zero-day exploits. Our system takes an approach that is quite different from the above work.

8 Limitation and Conclusion

The current system still has some limitations. For example, when some attack activities evade the system calls (although difficult, but possible), or the attack time span is much longer than the analyzed time period, the constructed SOIDG may not reflect the complete zero-day attack paths. In such cases, our system can only reveal partial of the paths.

This paper proposes to use Bayesian networks to identify the zero-day attack paths. For this purpose, a System Object Instance Dependency Graph is built to serve as the basis of Bayesian networks. By incorporating the intrusion evidence and computing the probabilities of objects being infected, the implemented system Pr0bA can successfully reveal the zero-day attack paths at run-time.

Disclaimer

This paper is not subject to copyright in the United States. Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

References

1. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. ACM Computing Surveys (CSUR), 2009.
2. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. ESORICS, 2003.
3. S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. IEEE S&P, 2006.
4. S. T. King, Z. M. Mao, D. G. Lucchetti, P. M. Chen. Enriching intrusion alerts through multi-host causality. NDSS, 2005.
5. Y. Zhai, P. Ning, J. Xu. Integrating IDS alert correlation and OS-Level dependency tracking. IEEE Intelligence and Security Informatics, 2006.
6. S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. Managing Cyber Threats, 2005.
7. P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. ACM CCS, 2002.
8. X. Ou, W. F. Boyer, and M. A. McQueen. A scalable approach to attack graph generation. ACM CCS, 2006.
9. X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. USENIX security, 2005.
10. J. Dai, X. Sun, and P. Liu. Patrol: Revealing zero-day attack paths through network-wide system object dependencies. ESORICS, 2013.
11. Symantec Report. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf
12. Wireshark. https://www.wireshark.org/.
13. Snort. https://www.snort.org/.
14. Tcpdump. http://www.tcpdump.org/.
15. Tripwire. http://www.tripwire.com/.
16. S. T. King, and P. M. Chen. Backtracking intrusions. ACM SIGOPS, 2003.
17. X. Xiong, X. Jia, and P. Liu. Shelf: Preserving business continuity and availability in an intrusion recovery system. ACSAC, 2009.
18. Nessus. http://www.tenable.com/products/nessus-vulnerability-scanner.
19. L. Wang, S. Jajodia, A. Singhal, and S. Noel. k-zero day safety: Measuring the security risk of networks against unknown attacks. ESORICS, 2010.
20. M. Albanese, S. Jajodia, A. Singhal, and L. Wang. An Efficient Approach to Assessing the Risk of Zero-Day Vulnerabilities. SECRYPT, 2013.
21. L. Wang, S. Jajodia, A. Singhal, P. Cheng, and S. Noel. k-Zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 2014.
22. P. Xie, J. H. Li, X. Ou, P. Liu, and R. Levy. Using Bayesian networks for cyber security analysis. DSN, 2010.

23. SamIam. http://reasoning.cs.ucla.edu/samiam/.
24. R. Tarjan. Depth-first search and linear graph algorithms. SIAM journal on computing 1, 1972.
25. GraphViz. http://www.graphviz.org/.
26. Ntop. http://www.ntop.org/.
27. L. Bilge, and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. ACM CCS, 2012.

Appendix

Table 5: System Call Dependency Rules

Dependency	Events	System calls
process→file	a process creates or writes a file	write, pwrite64, rename, mkdir, fchmod, chmod, fchownat, etc.
file→process	a process reads or executes a file	stat64, read, pread64, execve, etc.
process→process	a process creates or kill a process	vfork, fork, kill, etc.
process→socket	a process writes a socket	write, pwrite64, send, sendmsg, etc.
socket→process	a process reads a socket	read, pread64,recv, recvmsg, etc.
socket→socket	socket communication	sendmsg, recvmsg, etc.

Algorithm 1 Algorithm of SOIDG Generation

Require: set D of system object dependencies
Ensure: the SOIDG graph $G(V, E)$

for each *dep:* $src \rightarrow sink \in D$ **do**
  look up the most recent instance src_k of src, $sink_z$ of $sink$ in V
  if $sink_z \notin V$ **then**
    create new instances $sink_1$
    $V \leftarrow V \cup \{sink_1\}$
    if $src_k \notin V$ **then**
      create new instances src_1
      $V \leftarrow V \cup \{src_1\}$
      $E \leftarrow E \cup \{src_1 \rightarrow sink_1\}$
    else
      $E \leftarrow E \cup \{src_k \rightarrow sink_1\}$
    end if
  end if
  if $sink_z \in V$ **then**
    create new instance $sink_{z+1}$
    $V \leftarrow V \cup \{sink_{z+1}\}$
    $E \leftarrow E \cup \{sink_z \rightarrow sink_{z+1}\}$
    if $src_k \notin V$ **then**
      create new instances src_1
      $V \leftarrow V \cup \{src_1\}$
      $E \leftarrow E \cup \{src_1 \rightarrow sink_{z+1}\}$
    else
      $E \leftarrow E \cup \{src_k \rightarrow sink_{z+1}\}$
    end if
  end if
end for

Algorithm 2 Algorithm of Candidate Zero-day Attack Paths Identification

Require: the SOIDG graph $G(V, E)$, a vertex $v \in V$
Ensure: the candidate zero-day attack path $G_z(V_z, E_z)$

function $DFS(G, v, direction)$
  set v as *visited*
  if $direction = ancestor$ **then**
    set $next_v$ as parent of v that $next_v \rightarrow v \in E$
    set *flag* as *has_high_probability_ancestor*
  else if $direction = descendant$ **then**
    set $next_v$ as child of v that $v \rightarrow next_v \in E$
    set *flag* as *has_high_probability_descendant*
  end if
  for all $next_v$ of v **do**
    if $next_v$ is not labeled as *visited* **then**
      if the probability for $next_v$ $prob[next_v] \geq threshold$ **or** $next_v$ is marked as *flag* **then**
        set *find_high_probability* as $True$
      else
        $DFS(G, next_v, direction)$
      end if
    end if
    if *find_high_probability* is $True$ **then**
      mark v as *flag*
    end if
  end for
end function
for all $v \in E$ **do**
  $DFS(G, v, ancestor)$
  $DFS(G, v, descendant)$
end for
for all $v \in V$ **do**
  if $prob[v] \geq threshold$ **or** (v is marked as *has_high_probability_ancestor* **and** v is marked as *has_high_probability_descendant*) **then**
    $V_z \leftarrow V_z \cup v$
  end if
end for
for all $e : v \rightarrow w \in E$ **do**
  if $v \in V_z$ **and** $w \in V_z$ **then**
    $E_z \leftarrow E_z \cup e$
  end if
end for