Towards Probabilistic Identification of Zero-day Attack Paths

Xiaoyan Sun*†, Jun Dai†, Peng Liu*, Anoop Singhal‡ and John Yen*
*Pennsylvania State University, University Park, PA 16802, USA
†California State University, Sacramento, CA 95819, USA
‡National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
Email: {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 ZePro. An object instance graph is first built from system calls to capture the intrusion propagation. To further reveal the zero-day attack paths hiding in the instance graph, our system constructs an instance-graph-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 zero-day attack paths. The experiment results show that our system can effectively identify zero-day attack paths.**

I. Introduction

Defending against zero-day attacks is one of the most fundamentally challenging security 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 hard 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.

Considering the extreme difficulty of detecting individual zero-day exploits, a substantially more feasible strategy is to identify zero-day attack paths. In real world, to achive the attack goal, attack campaigns rely on a chain of attack actions, which forms an attack path. Each attack chain is a partial order of exploits and each exploit is exploiting a particular vulnerability. A *zero-day attack path* is a multi-step attack path that includes one or more zero-day exploits. A key insight in dealing with zero-day attack paths is to analyze the chaining effect. Typically, it is not very likely for a zero-day attack chain to be 100% zero-day, namely having every exploit in the chain be a zero-day exploit. Hence, defenders can assume that 1) the non-zero-day exploits in the chain are detectable; 2) these detectable exploits have certain chaining relationships with the zero-day exploits in the chain. As a result, connecting the detected non-zero-day segments through a path is an effective way of revealing the zero-day segments in the same chain.

Both alert correlation [4], [5] and attack graphs [6]–[9] are possible solutions for generating potential attack paths, but they are limited in revealing the zero-day ones. 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 all segments into a meaningful path and reveal the zero-day segments, especially when different segments may belong to totally irrelevant attack paths.

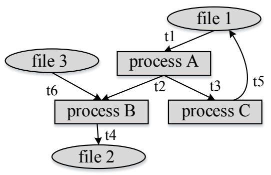
To address these limitations, Dai et al. proposed a system called Patrol [10] to identify real zero-day attack paths from a large set of suspicious intrusion propagation paths generated through tracking dependencies between OS-level objects. The set of suspicious dependency paths is usually very huge or even suffers from serious *path explosion* problem. A root cause for such explosion is that dependencies introduced by legitimate activities and dependencies introduced by zero-day attacks are often tangled together. Hence, Patrol made an assumption that extensive pre-knowledge are available to distinguish real zero-day attack paths from suspicious ones: common features or attack patterns of known exploitations can be extracted at the OS-level to help recognize future unknown exploitations if similar features appear again. However, this assumption is too strong in that 1) the acquirement of such pre-knowledge is quite difficult. It is a very ad hoc and effort consuming process. It relies heavily on the availability of the history for known vulnerability exploitations; 2) Even if the history is available, investigating and crafting the common features at OS-level for all types of exploitations requires immeasurable amount of human analysts' efforts or even the whole community's efforts.

Therefore, in this paper, we propose a probabilistic approach to identify the zero-day attack paths. Our approach is to 1) establish an *object instance graph* 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 instance graph 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 instance-graph-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 zero-day attack path. This approach does not require any pre-knowledge for the common features of known exploitations at OS-level.

The significance of our approach is as follows: 1) Our approach is systematic because Bayesian networks can incorporate literally all kinds of knowledge the defender has about the zero-day attack paths. The knowledge includes but is not limited to alerts generated by security sensors such as IDS and Tripwire, reports provided by vulnerability scanners, system logs, or even human inputs. 2) Our approach does not rely on particular assumptions or preconditions. Therefore, it is

```
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

Figure 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.

applicable to almost all kinds of enterprise networks. 3) Our approach is elastic. Whenever new knowledge is gained about zero-day attacks, such new knowledge can be incorporated and the effectiveness of our approach can be enhanced. Whenever erroneous knowledge is identified, our approach can easily get rid of the negative effects of the wrong knowledge. 4) The tool we built is automated. Today's security analysis relies largely on the manual work of human security analysts. Our automated tool can significantly save security analysts' time and address the human resource challenge.

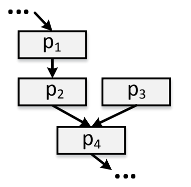
To summarize, we made the following contributions.

- 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 system object level by introducing the object instance graph.
- We have designed and implemented a system prototype named ZePro, which can effectively and automatically identify zero-day attack paths.

II. Rationales and Models

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 [14], [15] has explored constructing system level dependency graphs by parsing system call traces. This type of dependency graph is known as System Object Dependency Graphs (SODGs). 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 as used in previous work [14], [15] can be adopted to generate such dependencies by parsing system calls such as read, write, fork, send, recv, and so on. Figure 1b is an example SODG generated by parsing the simplified system call log shown in Figure 1a.



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

Figure 2: An example Bayesian network.

A. Why use Bayesian Network?

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). Figure 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 $P(p_1 = T|p_2 = T)$.

The BN can be applied on top of the system level dependency graph for the following benefits. First, BN is an effective tool to incorporate intrusion evidence from a variety of information sources. Alerts generated by different security sensors are usually isolated from each other. As a unified platform, BN is able to leverage these alerts as attack evidence to aid the security analysis. Second, BN can quantitatively compute the probabilities of objects being infected. The inferred probabilities are the key guidance to identify zero-day attack paths. By only focusing on the objects with high infection probabilities, the set of suspicious objects can be significantly narrowed down. The zero-day attack paths formed by the high-probability objects through dependency relations is thus of manageable size.

B. Problems of Constructing BN 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 Figure 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 Figure 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. Therefore, in this paper we propose a new type of dependency graph, the *object instance graph*, to address the above problems.

C. Object Instance Graph

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

Definition 1. *Object Instance Graph*
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 object instance graph 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: $src_i \rightarrow sink_j$, 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 $E = E \cup \{dep_s\text{: } sink_{j-1} \rightarrow sink_j\}$.
- If $a \rightarrow b \in E$ and $b \rightarrow c \in E$, then c transitively depends on a.

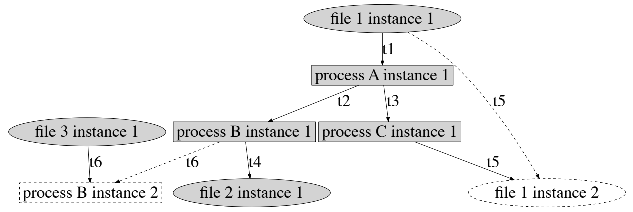


Figure 3: An instance graph generated by parsing the same set of simplified system call log as in Figure 1a. The label on each edge shows the time associated with the corresponding system call operation. The dotted rectangle and ellipse are new instances of already existed objects. The solid edges and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

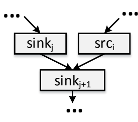
According to Definition 1, for src object, a new instance is created only when no instances of src exist in the instance graph. For $sink$ object, however, a new instance is created whenever a $src \rightarrow sink$ dependency appears. The underlying insight is that the status of the src object will not be altered by $src \rightarrow sink$, 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 and the new instance of the same object. 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 instance graph can well tackle the problems existing in the SODG for constructing BNs. It can be illustrated using Figure 3, an instance graph created for the same simplified system call log as in Figure 1a. First, the instance graph 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*, Figure 3 parsed it into *file 3 instance 1→process B instance 2*. Comparing to Figure 1b in which *file 3* has indirect infection causality on *file 2* through *process B*, the instance graph in Figure 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, instance graphs can break the cycles contained in SODGs. Again, in Figure 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 Figure 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.

III. Instance-graph-based Bayesian Networks

To build a BN based on an instance graph 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,



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	$1-\tau$	$1-\rho$

Figure 4: The infection propagation models.

evidence from different information sources has to be incorporated into BN for subsequent probability inference.

A. *The Infection Propagation Models*

In instance-graph-based BNs, each object instance has two possible states, *"infected"* and *"uninfected"*. Our infection propagation models 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 instance graphs.

Contact Infection Causality Model. This model captures the infection propagation between instances of two different objects. Contact infection causality is formed due to the information flow between the two objects in a system call operation. Figure 4 shows a portion of BN constructed when a dependency $src{\rightarrow}sink$ occurs and the CPT for $sink_{j+1}$. 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, $0 \le \tau, \rho \le 1$.

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.

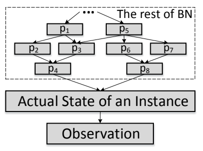
The contact infection rate τ determines how likely $sink_{j+1}$ gets infected when src_i is infected. The value of τ determines to which extent the infection can be propagated within the range of an instance graph. In an extreme case where $\tau = 1$, all the object instances will get contaminated as 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. We will evaluate the impact of τ and ρ in Section VI.

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"*[1]. 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 exemplified in Figure 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.

B. *Evidence Incorporation*

BN is able to incorporate security alerts from a variety of information sources as the evidence of attack occurrence.

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



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

Figure 5: Local observation model [19].

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 instance-graph-based BN by directly changing the infection status of the corresponding instance into *infected*. Second, leverage the local observation model (LOM) [19] 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 Figure 5, an observation node can be added as the direct child node to an object instance. The implicit causality 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, $P(\textit{Observation = True} \mid \textit{Actual = Uninfected})$ shows the false positive rate and $P(\textit{Observation = False} \mid \textit{Actual = Infected})$ indicates the false negative rate.

IV. System Design

Figure 6 shows the overall system design.

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.

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 instance graph can be constructed by concatenating individual host-wide instance graphs through instances of the communicating sockets.

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

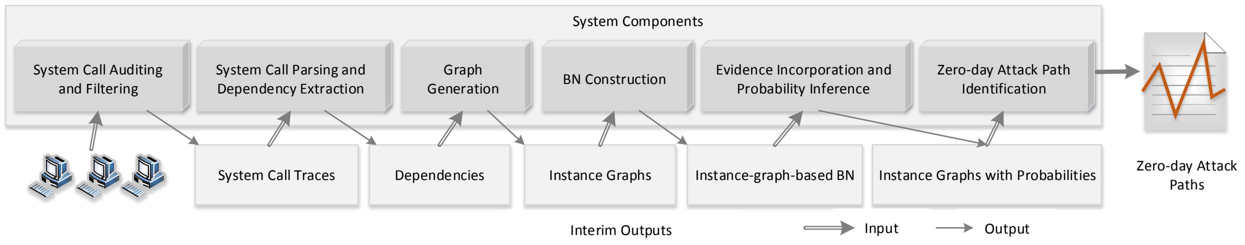

System Components
System Call Auditing and Filtering
System Call Parsing and Dependency Extraction
Graph Generation
BN Construction
Evidence Incorporation and Probability Inference
Zero-day Attack Path Identification
System Call Traces
Dependencies
Instance Graphs
Instance-graph-based BN
Instance Graphs with Probabilities
Zero-day Attack Paths
Interim Outputs
Input
Output

Figure 6: System design.

Zero-day attack paths identification. To reveal the zero-day attack paths from the mess of instance graphs, the nodes with high probabilities are to be preserved, while the link between them should not be broken. We implemented an algorithm based on the depth-first search (DFS) algorithm to tag each node in the instance graph 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.

V. Implementation

The whole system includes online system call auditing and off-line data analysis. System call auditing 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 instance-graph-based BN and a *dot*-compatible file for visualizing the zero-day attack paths in Graphviz [21], and 145 lines of Java code for probability inference, leveraging the API provided by the BN tool SamIam [20].

An instance graph can be very large due to the introduction of instances. Therefore, in addition to system call filtering, we also develop several ways to prune that instance graphs 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 prune an instance graph 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 Figure 3 only appears as the src object in the dependencies parsed from the system call log in Figure 1a, so *file 3 instance 1* can be ignored in the simplified instance graph. 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 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 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 instance graph 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.

VI. Experiments

A. Attack Scenario

To demonstrate the merits of our system and compare experiment results with Patrol [10], we implemented a similar attack scenario as in Patrol. We built a test-bed network and launched a three-step attack towards it. Figure 7 illustrates the attack scenario. Step 1, the attacker exploits vulnerability CVE-2008-0166 [12] 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 a vulnerability on a specific workstation. The public directory is shared among all the hosts in the test-bed network so that a workstation may access and download this malicious file. Step 3, once the malicious file is mounted and installed on the workstation, the attacker is able to execute arbitrary code on workstation. To verify the effectiveness of our approach, we conducted two major sets of experiments by providing different vulnerabilities in step 3. Due to space constraint, we only present the results for one set of experiment. In this experiment, the malicious file contains a Trojan-horse that exploits CVE-2009-2692 [11] existing in the Linux kernel of workstation 3.

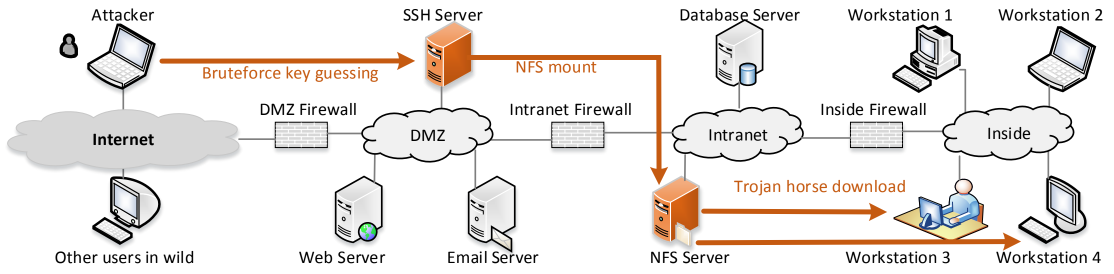


Figure 7: Attack scenario.

Since zero-day exploits are not readily available, we emulate zero-day vulnerabilities with known vulnerabilities. For example, we treat CVE-2009-2692 as zero-day vulnerabilities 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.

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 and Nessus. For sensors that need configuration, we tailored their rules or policy files to match our hosts.

B. Experiment Results

While simultaneously logging the system calls on each host and collecting the security alerts, we conducted the described three-step attacks. After analyzing a total number of 143120 system calls generated by three hosts, we constructed an instance-graph-based BN with 1853 nodes and 2249 edges.

Correctness. Given the evidence, Figure 8 illustrates the identified zero-day attack path in the form of instance graphs. The processes, files, and sockets are denoted with rectangles, ellipses, and diamonds respectively. The intrinsic infection rate ρ is set as 0.0001, and the probability threshold of recognizing high-probability nodes is 80%. The contact infection rates τ is respectively 0.9. We mark the evidence with red color and the nodes that are verified to be malicious with grey color. Figure 8 shows how the malicious file is uploaded from SSH server to NSF server, and then gets executed on workstation 3. Therefore, Figure 8 has testified the effectiveness of our approach for revealing actual zero-day attack paths.

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 3 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. Exposing the page-zero object can help system admins to further diagnose how the intrusion happens and propagates.

Table I: The Collected Evidence

ID	Host	Evidence
E1	SSH Server	Snort messages "potential SSH brute force attack"
E2	Workstation 3	Tripwire reports "/virus is added"
E3	Workstation 3	Tripwire reports "/etc/passwd is modified"
E4	Workstation 3	Tripwire reports "/etc/shadow is modified"

Size of Instance Graph and Zero-day Attack Paths. We also evaluated the size of instance graphs and the effectiveness of our pruning techniques for reducing the number of instances. Table II summarizes the impact of pruning instance graphs for each host. It shows that the number of instances is reduced from 39840 to 1853. On average each object has 2.03 instances, which is quite acceptable. To further gain the object-level comprehension of zero-day attack paths, ZePro also supports converting instance graphs to system object dependency graph by merging all the instances belonging to the same object into one node. Zero-day attack paths in SODG contain only objects and can be used for verification when details regarding instances are not needed. Figure 9 is the SODG form of zero-day attack paths for Figure 8.

The experiment results have demonstrated that our system ZePro substantially outperforms Patrol. Without any pre-knowledge towards known vulnerability exploits and OS-level exploitation features (which are mandatory information for Patrol to work), Zepro generates much better results than Patrol. In our experiment, the zero-day attack path identified by Patrol contains 175 objects, while the path by our system is composed of only 77 objects (Figure 9). Considering that the total number of objects involved in original instance graph is only 913, the 56% reduction of path size is substantial. More importantly, when the extensive pre-knowledge is not available (which is usual), ZePro remains as effective, but Patrol will result in a large number of suspicious intrusion propagation paths and is incapable of recognizing real attack paths hiding in these candidates. For example, in Patrol's dataset where SSH server takes a workload of 1 request per 5 seconds, a 15-minute system call log generates 180 candidate paths that tangle with the real zero-day attack paths.

Influence of Evidence. In our experiment, we choose a number of nodes in Figure 8 as the representative interested instances. Table III 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. In Table III, the results show that when

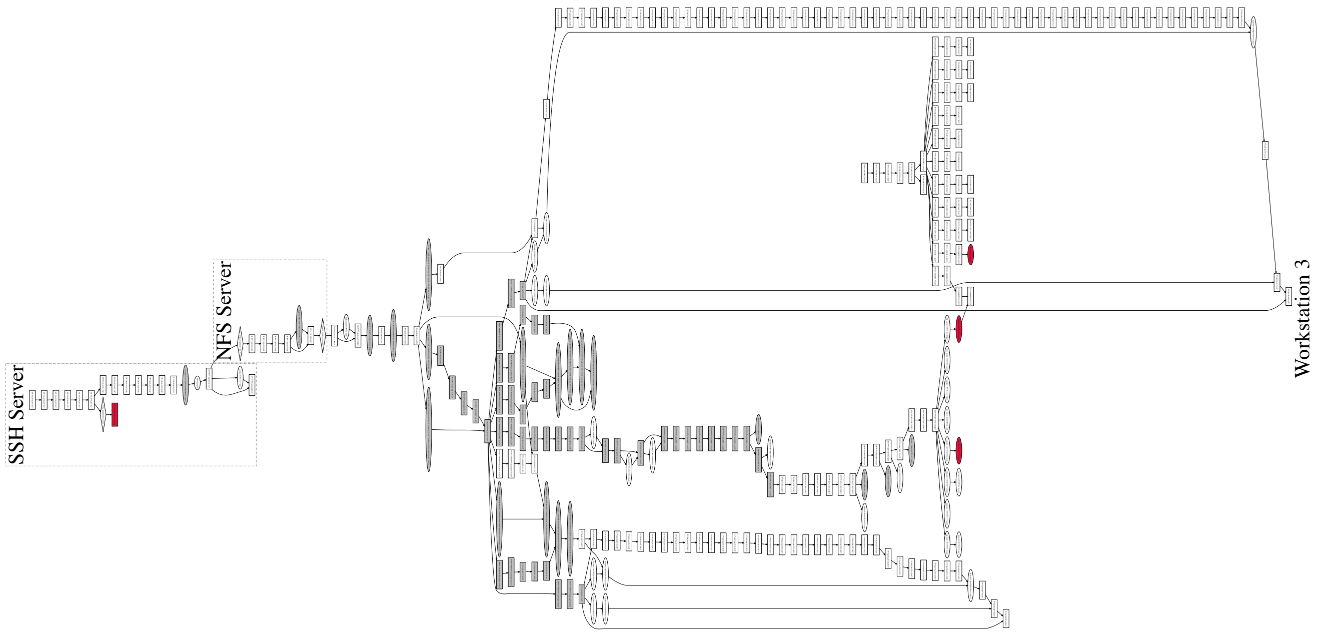


Figure 8: The zero-day attack path in the form of an instance graph.

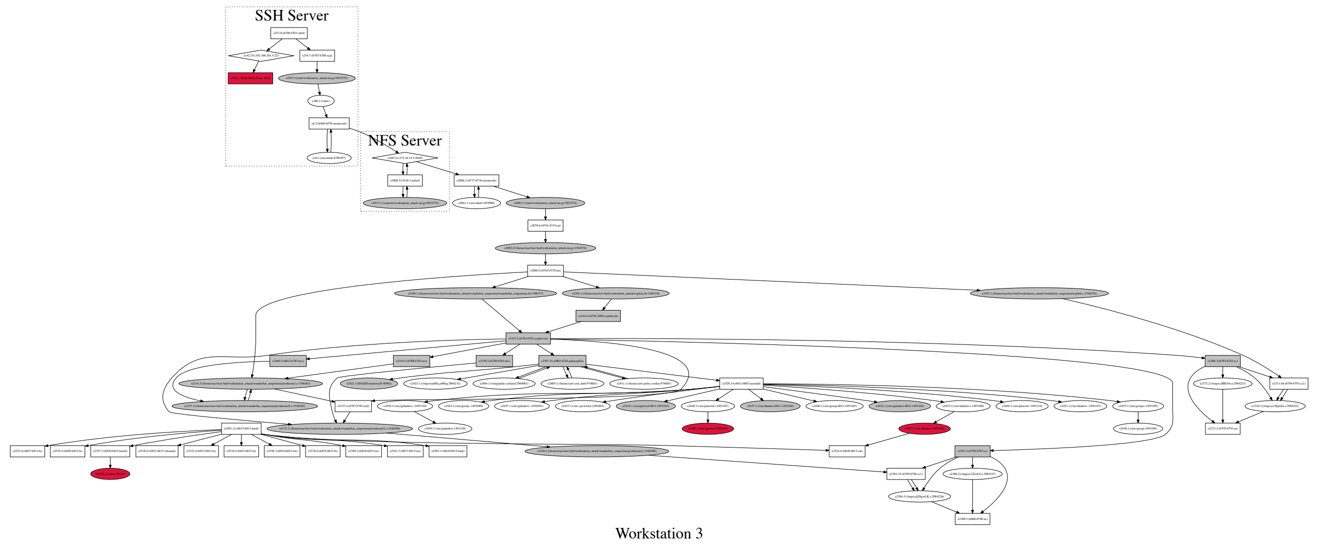


Figure 9: The object-level zero-day attack path.

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 graph. It is also possible that the arrival of some evidence may decrease 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.

Influence of False Alerts. We assume that *E4* is a false alarm generated by Tripwire and evaluate its influence to the BN output. Table IV 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 as more evidence is collected.

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

Table II: The Impact of Pruning the Instance Graphs

	SSH Server		***NFS Server***		***Workstation 3***	
	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 instance graph	10447	745	11544	39	17849	1069
number of dependencies(edges) in instance graph	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

Table III: The Influence of Evidence

Evidence	***SSH Server***			***NFS Server***		***Workstation 3***			
	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%

Table IV: 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%

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 and Scalability. We evaluated the time cost for off-line data analysis, which includes the time for instance-graph-based BN generation, BN 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 instance-graph-based BN generation and zero-day attack path identification, because the DFS algorithm is applied towards every node in the instance graph. For our experiments that conduct the off-line analysis on a host with 2.4 GHz Intel Core 2 Duo processor and 4G RAM, Table II shows the time required for constructing the instance-graph-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, the average data analysis speed is 280 KB/s, which is quite reasonable. The average memory used for compiling the BN is 4.32 Mb. As for the run-time performance overhead, the overall system slow-down caused by the system call logging component is around 15% to 20% according to the measurement with UnixBench and kernel compilation.

The scalability of the approach proposed in this paper can be ensured by the following aspects. First, the time window of collecting system call logs for analysis can be adjusted. For example, individual systems can collect system calls and send the logs to central machine for analysis every 30 or 40 minutes. In our experiments, a 40-minute system call log generates a BN with 1854 nodes. Smaller time window usually generates smaller BN size, but not always. The BN size mainly depends on the actual behavior of system call logs and cannot be estimated in a determined way. Second, although an enterprise network may contain a large number of hosts, the instance graphs generated by the individual hosts are not necessarily connected to each other. An actual network-wide instance graph often contains one or several isolated instance graphs. This also limits the size of individual BNs. Third, both instance graph generation and zero-day attack path identification can be conducted with parallel computing. Taking the current experiment results for estimation, if an enterprise network contains 10000 hosts and an analysis cluster with 512 processors, the time for instance graph generation and zero-day attack path identification could be 2.93 minutes and 6.3 minutes respectively. In addition, intensive research has been conducted towards the scalability of BN compilation and probability inference [23], [24]. A scalable parallel implementation using junction tree has been developed for exact inference in BN [24]. The recursive conditioning [25] algorithm we employed in this paper even offers a smooth tradeoff between time and space, which also enhances the scalability of BN inference.

VII. Related Work

The work that is most related to us is Patrol [10]. It touches the zero-day attack path problem at operating system level. Our work also targets the same problem, but is substantially different from Patrol in several aspects. First, Patrol relies on extensive pre-knowledge regarding known vulnerability exploitations to distinguish zero-day attack paths from the huge number of candidate paths. However, such pre-knowledge is extremely difficult to acquire and may not be useful when zero-day exploits do not share common features with previous exploits at OS-level. Instead, our approach does not require any pre-knowledge, and solely rely on collected intrusion evidence. Second, Patrol only conducts qualitative analysis and treats every object on the identified paths as having the same malicious status. Compared to Patrol, our approach quantifies the infection status of each system object with probabilities. By only focusing on system objects with relatively high probabilities, the set of suspicious objects can be significantly narrowed down and the size of revealed zero-day attack path is relatively small. Third, Patrol performs reachability analysis through tracking and thus generates a huge candidate pool for zero-day attack paths. In contrast, our system does not conduct tracking, but relies on the computed probabilities. The paths containing highly suspicious objects reveal themselves automatically. The dependency paths introduced by legitimate activities and the dependency paths introduced by zero-day attacks are therefore separated with ease.

Other related work includes system call dependency tracking and zero-day attack identification. System call dependency tracking is first proposed in [14] 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 computes the infection probabilities of system objects. [22] 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 [17], [18]. 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.

VIII. Limitation and Conclusion

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

In conclusion, this paper proposes to use Bayesian networks to identify the zero-day attack paths. For this purpose, an object instance 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 ZePro can successfully reveal the zero-day attack paths.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. We also thank Dr. Adnan Darwiche for publicly sharing SamIam and Arthur Choi for the help in SamIam Usage. This work was supported by ARO W911NF-15-1-0576, ARO W911NF-13-1-0421 (MURI), CNS-1422594, and NIETP CAE Cybersecurity Grant.

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] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692

[12] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166

[13] Symantec Report. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf

[14] S. T. King, and P. M. Chen. *Backtracking intrusions.* ACM SIGOPS, 2003.

[15] X. Xiong, X. Jia, and P. Liu. *Shelf: Preserving business continuity and availability in an intrusion recovery system.* ACSAC, 2009.

[16] Nessus. http://www.tenable.com/products/nessus-vulnerability-scanner.

[17] L. Wang, S. Jajodia, A. Singhal, S. Noel. *k-zero day safety: Measuring the security risk of networks against unknown attacks.* ESORICS, 2010.

[18] M. Albanese, S. Jajodia, A. Singhal, L. Wang. *An Efficient Approach to Assessing the Risk of Zero-Day Vulnerabilities.* SECRYPT, 2013.

[19] P. Xie, J. H. Li, X. Ou, P. Liu, and R. Levy. *Using Bayesian networks for cyber security analysis.* DSN, 2010.

[20] SamIam. http://reasoning.cs.ucla.edu/samiam/.

[21] GraphViz. http://www.graphviz.org/.

[22] L. Bilge, and T. Dumitras. *Before we knew it: an empirical study of zero-day attacks in the real world.* ACM CCS, 2012.

[23] Ole J. Mengshoel. *Understanding the scalability of Bayesian network inference using clique tree growth curves.* Artificial Intelligence 174.12 (2010): 984-1006.

[24] V. Krishna Namasivayam, V. K. Prasanna. *Scalable parallel implementation of exact inference in Bayesian networks.* ICPADS, 2006.

[25] Adnan Darwiche. *Recursive conditioning.* Artificial Intelligence 126. 1 (2001): 5-41.