Using Bayesian Networks for Probabilistic Identification of Zero-Day Attack Paths

Xiaoyan Sun, Jun Dai, Peng Liu, Anoop Singhal, and John Yen

***Abstract*— Enforcing a variety of security measures (such as intrusion detection systems, and so on) can provide a certain level of protection to computer networks. However, such security practices often fall short in face of zero-day attacks. Due to the information asymmetry between attackers and defenders, detecting zero-day attacks remains a challenge. Instead of targeting individual zero-day exploits, revealing them on an attack path is a substantially more feasible strategy. Such attack paths that go through one or more zero-day exploits are called zero-day attack paths. In this paper, we propose a probabilistic approach and implement a prototype system ZePro for zero-day attack path identification. In our approach, a zero-day attack path is essentially a graph. To capture the zero-day attack, a dependency graph named object instance graph is first built as a supergraph by analyzing system calls. To further reveal the zero-day attack paths hidden in the supergraph, our system builds a Bayesian network based upon the instance graph. By taking intrusion evidence as input, the Bayesian network is able to compute the probabilities of object instances being infected. Connecting the high-probability-instances through dependency relations forms a path, which is the zero-day attack path. The experiment results demonstrate the effectiveness of ZePro for zero-day attack path identification.**

***Index Terms*— Intrusion detection, network security, computer security, probability, Bayesian networks, system call, zero-day attack.**

I. Introduction

SECURING computer networks is gaining ever increasing importance. A security attack can potentially impact not only traditional enterprise networks, but also critical infrastructures that are controlled via networks. However, one major challenge of defending computer networks is the lack of means for detecting zero-day attacks, as "you cannot protect against what you do not know". Zero-day attacks usually exploit vulnerabilities that are unknown to public, including network defenders. The information asymmetry between attackers and defenders makes detection of zero-day attacks extremely difficult. Considering such extreme difficulty of detecting individual zero-day attacks, identifying an attack path containing the zero-day exploits is a substantially more feasible strategy.

A. Zero-Day Attack Path

Today's computer networks are usually deployed with basic security defense measures, such as firewall and IDSs (Intrusion Detection Systems). It's usually not easy for attackers to directly break into the target machine. Therefore, attackers often rely on a chain of attack actions to achieve the attack goal. Each attack chain is essentially a sequence of exploits, which forms an attack path. An exploit enabled by an unknown vulnerability is regarded as a zero-day exploit. If any of the exploits on an attack path is zero-day, the path becomes a *zero-day attack path*.

B. Key Insight

Given that a zero-day attack path is inherently a chain of attack actions, a key insight to deal with zero-day attack paths is to analyze the chaining effects. Generally, it is very unlikely for an attack path to be 100% zero-day. That is, since zero-day exploits are not readily available, it's very difficult for attackers to ensure that all exploits on an attack path are zero-day exploits. Therefore, a zero-day attack path is usually composed of both zero-day exploits and non-zero-day exploits. Consequently, defenders can make the following assumptions: 1) the non-zero-day exploits in the chain can be detected in some way, such as through the security sensors; 2) the detectable non-zero-day exploits have certain chaining relationships with the zero-day exploits in the chain. Hence, connecting the detected non-zero-day segments through a path is an effective approach to reveal the zero-day segments in the same chain.

To leverage the chaining effects, a critical step is constructing an appropriate graph that contains the chain. Therefore, in our work, we propose to firstly build a network wide supergraph from system calls, and then identify the zero-day attack paths hidden in the supergraph. The identified zero-day attack path is essentially a subgraph of the network wide supergraph. This approach is proposed due to four rationales. First, system calls are the only way for user programs to interact with kernel operating systems, and are thus hard-to-avoid and attack neutral. Attack neutral means that system calls can capture both legitimate and malicious system activities. Second, a network wide supergraph can be built by analyzing the collected system calls. The supergraph is also attack neutral. The zero-day exploits, if exist, will expose themselves in the graph. Third, the supergraph is essentially a set of paths. Zero-day attack paths are subsets of the supergraph, and can be identified in a certain way. Fourth, the zero-day attack paths

Manuscript received November 17, 2017; revised March 4, 2018; accepted March 19, 2018. Date of publication March 29, 2018; date of current version May 9, 2018. This work was supported in part by ARO under Grant W911NF-15-1-0576, Grant W911NF-13-1-0421 (MURI), and Grant CNS-1422594, and in part by NIST under Grant 60NANB17D279. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Issa Traore. *(Corresponding author: Xiaoyan Sun.)*

X. Sun and J. Dai are with the Department of Computer Science, California State University, Sacramento, CA 95819 USA (e-mail: xiaoyan.sun@csus.edu; jun.dai@csus.edu).

P. Liu and J. Yen are with the College of Information Sciences and Technology, Penn State University, University Park, PA 16802 USA (e-mail: pliu@ist.psu.edu; jyen@ist.psu.edu).

A. Singhal is with the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA (e-mail: anoop.singhal@nist.gov).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2821095

provide a network-wide attack context for recognizing zero-day exploits. With the path, the accuracy and performance of detecting zero-day exploits is better than the detection within the isolated per-host context.

C. Patrol

Following the philosophy of "building a network wide supergraph and then identifying the zero-day attack paths hidden in it", we first developed a prototype system called Patrol [1]. The general idea of Patrol is as follows: 1) By analyzing system calls, a network-wide system object dependency graph (SODG) is generated as the supergraph. The SODG model was invented in pioneering work [3] and formalized in [1]. Details about SODG are provided in Section II-A. Since system calls capture both legitimate and malicious system activities, a SODG reflects the infection propagation processes through the dependencies among system objects, such as processes, files and sockets; 2) Given the SODG, a number of suspicious intrusion propagation paths (SIPPs) are identified through backward and forward tracking from some trigger nodes, which are usually suspicious system objects that are involved in security alerts; 3) The suspicious paths are verified whether they are true zero-day attack paths through policy checking. Patrol demonstrated the effectiveness of our key insight and was able to successfully identify zero-day attack paths. However, it also suffers from serious path explosion problem. That is, when a large number of security alerts are available, the tracking mechanism in Patrol can result in too many suspicious paths. In Patrol, distinguishing true zero day attack paths from suspicious ones relies on extensive pre-knowledge for the common features of known exploitations at OS-level, which is not readily available. Moreover, the size of suspicious paths can be very large as Patrol preserves every tracking-reachable objects. Therefore, the deterministic dependency analysis is not adequate and will fall short.

D. ZePro

In this paper, we propose a new probabilistic approach for zero-day attack path identification. This newly developed system, named ZePro, follows the same philosophy as Patrol, but with substantial difference. First, instead of using SODG, the new approach establishes an *object instance graph* as the supergraph. Each instance is a "version" of a system object at a specific time. With instances, the object instance graph can capture the state changes of an object, and the root cause for these changes. Second, instead of tracking from trigger nodes, the new approach builds a Bayesian network (BN) based on the instance graph to leverage intrusion evidence. Fed with evidence, the BN can quantitatively infer the probabilities of object instances being infected. By connecting instances with high infection probabilities, a path can be formed. This path is the zero-day attack path.

Our approach has the following significance. First, this approach does not rely on particular OS-level pre-knowledge. In Patrol, it has to firstly extract OS-level common features from known exploitations. These features are then used as the pre-knowledge to distinguish true zero-day attack paths from suspicious ones. The effectiveness of Patrol largely depends on the availability of the common features. However, our approach in ZePro remains applicable and effective even when such pre-knowledge is not available. Second, this approach leverages the intrusion evidence in a systematic way. BN can incorporate literally all varieties of intrusion evidence, such as security alerts, vulnerability scanning results, system logs, or even human knowledge. Rather than generating individual suspicious intrusion propagation paths through tracking, all intrusion evidence fed into BN generates synthesized impact towards the infection probabilities of object instances. Third, this approach is elastic. New intrusion evidence can be incorporated as it is collected. The new evidence may change the previous probability inference results. Moreover, erroneous knowledge will be ruled out as more true evidence is fed into BN. Fourth, the tool ZePro is automated, which greatly enhances security analysts' working effectiveness and efficiency.

This paper is developed based on our continuous work [1] and [2], where [2] is an innovation to its predecessor work [1]. The most significant extensions made in this paper include more rationales on model evolution (Section II), more design specifics of the system modules (Section IV) including the algorithms of generating the object instance graphs (Section IV-C) and constructing the instance-graph-based Bayesian networks (Section IV-F), more implementation details (Section V), another experiment comprising a new attack (Section VI-A), a more thorough analysis of experiment results (Section VI-B), and a more comprehensive review of related work (Section VII).

The contribution of this paper is summarized as follows.

- To the best of our knowledge, ZePro is the first work taking a probabilistic approach towards zero-day attack path identification.
- We proposed the methodology of building a network-wide supergraph and then identifying the zero-day attack paths hidden in it.
- We made the first effort to construct Bayesian networks at OS level by introducing the object instance graph.
- We designed and implemented the system prototype ZePro, which can effectively and automatically identify zero-day attack paths.

II. Model Evolution

A. Predecessor Model

To reveal the zero-day attack paths in an enterprise-level network, a model was proposed and constructed in [1], which was the first work noticing the zero-day attack path problem and also the first attempt to address this new issue. The model assumes that a typical enterprise network consists of mainly Unix-like operating systems (OS), in which the system entities can be categorized into processes, files and sockets.

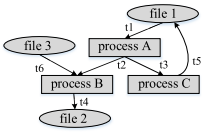
1) System Object Dependency Graph: By analyzing system calls, a System Object Dependency Graph can be constructed as the supergraph to capture the intrusion propagation.

Definition 1 (System Object Dependency Graph (SODG) [1]): If the system call trace for the i-*th* host is denoted as Σ_i, then the SODG for the host is a directed graph $G(V_i, E_i)$, where:

- V_i is the set of nodes, initialized to empty set $\varnothing$; and E_i is the set of directed edges, initialized to empty set $\varnothing$;
- If a system call $syscall \in \Sigma_i$, and *dep* is the dependency relation parsed from *syscall* according to dependency rules in [1], where $dep \in \{(src \rightarrow sink), (src \leftarrow sink), (src \leftrightarrow sink)\}$, *src* and *sink* are OS objects (mainly a process, file or socket), then $V_i = V_i \cup \{src, sink\}$,

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

(a)



(b)

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. (a) Simplified system call log in timeorder. (b) SODG.

$E_i = E_i \cup \{dep\}$. *dep* inherits timestamps *start* and *end* from *syscall*;

- If $(a \rightarrow b) \in E_i$ and $(b \rightarrow c) \in E_i$, then c transitively depends on a.

As defined in Definition 1, system calls are parsed into system objects and dependencies between them. For example, system call *sys_read* infers a dependency relation from the accessed file to the reading process (*file→process*); system call *sys_write* determines a dependency from the writing process to the accessed file (*process→file*). The dependency rules to parse the system calls are provided in [1]. Based on these rules, the system calls of interest are parsed into three parts: a *src* object, a *sink* object, and a *dep* relation between them. The unions of such objects and depencency relations inherently form a directed graph. Figure 1b illustrates an example SODG resulted from the system call log shown in Figure 1a.

The major merit of SODG model is its embodiment of the *infection propagation* process. Usually an infection is manifested by one or more objects in the victim system that are created or tainted by the intrusion attacks. Such objects can be a misled process, a trojan file or corrupted data. System calls that access these seed objects may cause the infection propagate to other innocent objects along the outgoing direct or transitive dependency relations, throughout the system or even across the network. Such infection propagation will reflect all exploits including zero-day ones, as system calls are attack-neutral and hard to avoid. In other words, SODG, inherently a set of paths, will neutrally capture the zero-day attack paths as long as one exists.

2) Suspicious Intrusion Propagation Path: The previous works [1], [3], [4] have explored the various ways to exploit SODG, e.g. for backtracking or quarantining intrusions. The philosophy taken by [1] is to dig out paths revealing the intrusion break-ins and its cascading malicious activities, namely Suspicious Intrusion Propagation Paths (SIPPs), from the SODG.

Definition 2 (Suspicious Intrusion Propagation Paths (SIPPs) [1]): If the network-wide SODG is denoted as $\cup G(V_i, E_i)$, where $G(V_i, E_i)$ denotes the per-host SODG for the i-*th* host, then the SIPPs are a subgraph of $\cup G(V_i, E_i)$, denoted as $G(V', E')$, where:

- V' is the set of nodes, and $V' \subset \cup V_i$;
- E' is the set of directed edges, and $E' \subset \cup E_i$;
- V' is initialized to include *trigger nodes* only;
- For $\forall obj' \in V'$, if $\exists obj \in \cup V_i$ where $(obj \rightarrow obj') \in \cup E_i$ and $start(obj \rightarrow obj') \leq lat(obj')$, then $V' = V' \cup \{obj\}$ and $E' = E' \cup \{(obj \rightarrow obj')\}$. $lat(obj')$ maintains the *latest access time* to obj' by edges in E';
- For $\forall obj' \in V'$, if $\exists obj \in \cup V_i$ where $(obj' \rightarrow obj) \in \cup E_i$ and $end(obj' \rightarrow obj) \geq eat(obj')$, then $V' = V' \cup \{obj\}$ and $E' = E' \cup \{(obj' \rightarrow obj)\}$. $eat(obj')$ maintains the *earliest access time* to obj' by edges in E'.

As defined in Definition 2, the SIPPs is by nature a subset (also a subgraph) of SODG. A *trigger node* is used to dig out the SIPPs from SODG: the objects that constitute an SIPP are the ones that either have affected the trigger node through direct or transitive dependency relations before its latest access time, or have been affected by the trigger node after its earliest access time. Trigger nodes can be OS objects that are revealed by security sensors, such as an irregularly communicating network socket recognized by Snort [5], or a modified file noticed by Tripwire [6].

Taking Figure 1b for example, if *file 3* is a trigger node, *file 3→process B→file 2* can be an SIPP. A network-wide SODG can be unmanageably complex and thus difficult for human analysts to understand. Therefore, extracting SIPPs from SODGs can effectively narrow down the suspect objects attributed to infection propagation. An SIPP is usually much smaller in size. As illustrated in [1], A 15-minute system call log with *143,120* system calls generated a 3-host SODG with *1,288* objects and *50,519* dependencies. In contrast, an SIPP identified from this SODG contains only *175* objects.

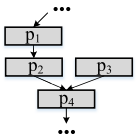
The SIPPs neutrally captures almost all candidate zero-day attack paths. The only chance for a zero-day attack path to escape from SIPPs is that all exploits on the path are zero-day and no security sensors are triggered. This is very unlikely, as it's very difficult for attackers to exploit only zero-day vulnerabilities along the path and ensure all zero-day exploits undetected by security sensors. As long as one known vulnerability is exploited or any security sensors are triggered, the attack path will be captured as SIPPs.

3) Path Explosion: Through tracking dependencies between OS objects, the system Patrol [1] can build network-wide SODGs, dig out SIPPs, and further identify the real zero-day attack paths. However, Patrol often suffers from serious *path explosion* problem because the set of SIPPs can be very large. 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 is available to distinguish real zero-day attack paths from suspicious ones. The pre-knowledge are common features or attack patterns of known exploitations that are extracted at the OS-level to help recognize future unknown exploitations. If a similar feature is detected on an SIPP, it indicates the existence of a zero-day exploit. However, this assumption is too strong in that:

- 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;
- 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.

B. Successor Model

To overcome the aforementioned problem, we need an approach that does not rely on the existence of any pre-knowledge for the common features or patterns of known exploitations at OS-level. Therefore, in this paper, we propose a probabilistic approach, which uses Bayesian networks to fuse intrusion evidence and requires no OS-level pre-knowledge.



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.

1) Why Use Bayesian Networks: The Bayesian networks (BN) is a probabilistic graphical model that embodies the cause-and-effect relations. It is by definition a Directed Acyclic Graph (DAG) that consists of a set of nodes and directed edges, where a node represents a variable of interest, and an edge represents the causality relation between two nodes. The causality relation is carried by a node-specific conditional probability table (CPT). Figure 2 gives an example BN and the CPT associated with p_2, according to which one can infer that the probability of p_2 being true is 0.9 when p_1 is true. Such conditional probability is denoted as $P(p_2 = T|p_1 = T) = 0.9$. Moreover, BN is also capable of incorporating evidence to update the posterior probabilities of variables of interest. For example, after evidence $p_2 = T$ is observed, it can be incorporated to update the probability of p_1 by computing $P(p_1 = T|p_2 = T)$.

Due to BN's capability of modeling cause-and-effect relations, it can be applied upon system-level dependency graphs. The rationale is that an affinity exists between BN and system-level dependency graphs: the dependency relations between system objects in system-level dependency graphs imply the infection causalities in BNs. For example, a dependency *process*→*file* in a dependency graph can become an infection causality relation in BN: *file* is likely to be infected if *process* is already infected. In this way a BN can be constructed based on the structure topology of a system-level dependency graph. The dependency-graph-based BN intrinsically enables the following benefits:

- It effectively incorporates intrusion evidence from a variety of information sources. The intrusion alerts are usually scattered here and there from apart security sensors. As a unified platform, the dependency-based BN leverages such alerts as attack evidence to facilitate security diagnosis;
- It quantitatively computes the probabilities of objects being infected. The inferred probabilities guide the identification of zero-day attack paths: the objects with high infection probabilities and the dependencies that carry the probabilistic inferences make candidate zero-day attack path stand out.

As a type of system-level dependency graph, SODG has the possibility of being the base graph for BN. However, constructing BN based on SODG is problematic due to the following reasons.

First, as a graphical model with only nodes and directed edges, BN can only inherit the structural information from SODG and cannot preserve the time labels associated with edges. Lack of time information will cause incorrect causality inference in the SODG-based BN. 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, but BN is an *acyclic* model. For instance, *file 1*, *process A* and *process C* in Figure 1b form a cycle. The cycle will be inevitably inherited from the SODG into the SODG-based BN. However, BN is supposed to be *acyclic* and thus does not allow any cycles.

2) Object Instance Graph: To address the above issues, we propose a new type of dependency graph, *object instance graph*. In object instance graph, a node is no longer an object. Rather, it is an instance of the object with a specific timestamp. Different instances are different "versions" of the same object at different time points, and thus they can have different infection status. Compared with the SODG generated from the same system call log, the object instance graph has equal or bigger size. We will address the scalability of our approach in Section VI-C.4. The object instance graph is defined as follows.

Definition 3 (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, initialized to empty set $\varnothing$; and E is the set of directed edges, 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 $E = E \cup \{dep_s: sink_{j-1} \rightarrow sink_j\}$.
- If $a \rightarrow b \in E$ and $b \rightarrow c \in E$, then c transitively depends on a.

According to Definition 3, for src object, a new instance is created only when no instance of src exists in the instance graph; for $sink$ object, however, a new instance is created whenever a $src \rightarrow sink$ dependency appears. The rationale is that the status of the src object will not be altered by $src \rightarrow sink$, while the status of $sink$ will be influenced. This dependency also causes an edge dep_c added between the most recent instance of src and the newly created instance of $sink$, as well as an edge dep_s added between the most recent instance and the new instance of the same object. We name dep_c as *contact dependency* as it is generated by the *contact* between two different objects via system call, and name dep_s as *state transition dependency* because it is caused by the state transition between different instances of the same system object. It is essential and reasonable to have dep_s because the status of the new instance of one object can be influenced

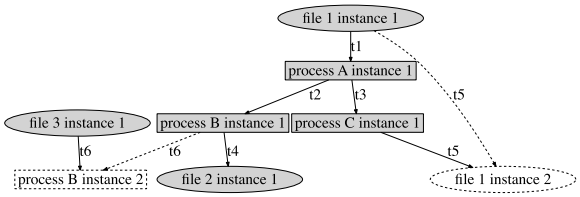


Fig. 3. An instance graph generated by parsing the same set of simplified system call log as in Figure 1a. The label on each edge shows the time associated with the corresponding system call operation. The dotted rectangle and ellipse are new instances of already existing objects. The solid edges and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

by the status of its most recent instance. Figure 3 illustrates an instance graph created for the same simplified system call log as in Figure 1a. It's a good example to show that the instance graph can well tackle the problems of SODG for BN construction.

First, by implying time information stamped onto specific instances, the instance graph is able to reflect correct infection causality relations. 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*. Compared 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 Figure 3 *file 3* does not have infection causality on *file 2*, while in Figure 1b it does.

Second, the instance graph can break the cycles contained in SODG. For example, 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. Hence, instead of pointing back to *file 1*, the edge from process C is directed to a new instance of *file 1*, which breaks the cycle formed by *file 1, process A* and *process C* in Figure 1b.

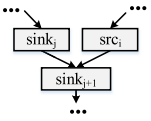
III. Instance-Graph-Based Bayesian Networks

Two steps are prescribed to build an instance-graph-based BN and compute probabilities for variables of interest: 1) the CPT tables have to be specified for each node via constructing proper infection propagation models; 2) evidence from different information sources has to be incorporated into BN for subsequent probability inference.

A. Constructing the Infection Propagation Models

In an instance-graph-based BN, each object instance is either *"infected"* or *"uninfected"*. The infection propagation models handle two types of infection causalities: *contact infection causalities* and *state transition infection causalities*, which respectively corresponds to the *contact dependencies* and *state transition dependencies* in Definition 3.

1) 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 via 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, an *intrinsic infection rate* ρ and a *contact infection rate* τ, $0 \leq \rho, \tau \leq 1$.



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.

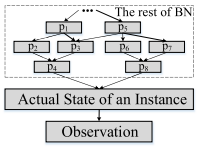
The intrinsic infection rate ρ decides how likely $sink_{j+1}$ gets infected when src_i is uninfected. In this case, since src_i is not the infection source of $sink_{j+1}$, if $sink_{j+1}$ is infected, it should be caused by other factors. So ρ can be determined by the prior probabilities of an object 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 the extent to which the infection can be propagated within an instance graph. In an extreme case where $\tau = 1$, all the object instances which have contact with the infected objects will get contaminated. In contrast, when $\tau = 0$ the infection will be confined inside the infected object and does not propagate to any other contacting object instances. Our system allows the value of τ tuned based on knowledge and experience. The impacts of τ and ρ are evaluated in Section VI-C.3.

2) State Transition Infection Causality Model: This model captures the infection propagation between instances of the same objects. The underlying rule is that, once *"infected"*, an object can never return to the state of *"uninfected"* (we make an assumption that no intrusion recovery operations are conducted). That is, once an instance of an object get infected, all future instances will keep the same infected state, regardless of the infection status of other contacting object instances. This is enforced by the CPT exemplified in Figure 4: if $sink_j$ is infected, the infection probability of $sink_{j+1}$ remains as 1, no matter whether src_i is infected or not; if $sink_j$ is uninfected, the infection probability of $sink_{j+1}$ is determined by the infection status of src_i according to the contact infection causality model.

B. Incorporating Evidence

The instance-graph-based BN incorporates alerts from a variety of security sensors as the evidence of attack occurrence via two ways: 1) evidence can be fed into BN by directly labeling the corresponding object instance as *infected*. If an object is scrutinized and confirmed to be already infected, the infection status of the corresponding instance at that specific time should be updated to be *infected*; 2) a type of nodes, namely Local Observation Model (LOM) nodes [95], can be added to BN to model the uncertainty towards observations. Observations mean the suspicious activities noticed by security sensors or admins that imply attack occurrences. The observations may contain uncertainty, such as suffering from false rates. In cases when observations are from security alerts, the CPT actually implies false rates of the particular security sensor: $P(Observation = True \mid Actual = Uninfected)$ shows the *false positive rate* and $P(Observation = False \mid Actual = Infected)$ indicates the *false negative rate*. As illustrated in Figure 5, an LOM node can be added as the direct *child* node to an object instance (i.e. *parent*). The observation



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 [95].

can be used to compute the posterior probability of its parent instances.

IV. System Design

Figure 6 shows the overall system design: we first perform system call auditing on each individual host, and then collect the system call traces to a central analysis machine for off-line instance-graph-based BN construction and zero-day path identification. The modules are specified as follows.

A. System Call Auditing and Filtering

System call is audited towards all running processes on each individual host. The reason is that it's hard to predict which process is involved in attacks, and process-confined system call auditing could miss intrusion-related system calls. The following information should be preserved during system call auditing: 1) system calls that contribute to socket communications, as they will be the glue to concatenate per-host instance graphs; 2) OS-aware information for accurate OS object identification, such as file absolute pathnames and inode numbers (merely file descriptor numbers is not sufficient due to reuse); 3) timestamp for each system call, as time information is critical to determine whether a system call and its relevant object instances are involved in infection propagation. To minimize performance degradation, system call auditing is the only system module that runs on the fly.

System call filtering is performed to reduce the bandwidth and CPU cost incurred by unfiltered data. System calls that involve highly redundant or possibly innocent objects should be filtered. Examples include the dynamic linked library files such as *libc.so.*$*$ and *libm.so.*$*$, dummy objects such as *stdin/stdout* and */dev/null*, pseudo-terminal master and slave (*/dev/ptmx* and */dev/pts*), log relevant objects such as *syslogd* and */var/log/*$*$, and objects relevant with system maintenance (*apt-get* and *apt-config*). The filtering can boost the speed of graph generation and path identification with reduced complexity.

B. System Call Parsing and Dependency Extraction

This module parses system calls into OS object instances and dependency relations between them, according to Definition 3. In addition, system call parameters also contribute to the parsing. They are used to uniquely recognize and name the nodes, and help infer the edge direction between them. For example, system call "*sys_open, start:470880, end:494338, pid:6707, pname:scp, pathname:/mnt/trojan, inode:9453574*" from our trace is transformed to (*6707, scp*)$\leftarrow$(*/mnt/trojan, 9453574*), where *pid* and *pname* are used to recognize the process, and *pathname* and *inode* are used to identify the file. The information flow direction is usually from file to process in system call *sys_open*, however the flag *O_CREAT* can invert the flow if it's observed in parameters.

Algorithm 1 Algorithm of Object Instance Graph Generation

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

```
for each dep: src→sink ∈D do
    look up the most recent instance src_k of src, sink_z of sink in V
    if sink_z∉V then
        create new instances sink_1
        V ← V ∪ {sink_1}
        if src_k∉V then
            create new instances src_1
            V ← V ∪ {src_1}
            E ← E ∪ {src_1→sink_1}
        else
            E ← E ∪ {src_k→sink_1}
        end if
    end if
    if sink_z∈V then
        create new instance sink_{z+1}
        V ← V ∪ {sink_{z+1}}
        E ← E ∪ {sink_z→sink_{z+1}}
        if src_k∉V then
            create new instances src_1
            V ← V ∪ {src_1}
            E ← E ∪ {src_1→sink_{z+1}}
        else
            E ← E ∪ {src_k→sink_{z+1}}
        end if
    end if
end for
```

C. Graph Generation

Resulted from system call parsing, the object instances and dependency relations become respectively the nodes and directed edges. The process of generating the object instance graph from system object dependencies is given in Algorithm 1, which strictly follows Definition 3. The constructed instance graph can be host-wide or network-wide, where the network-wide graph is constructed by concatenating individual host-wide ones. If and only if there exists at least one edge between any two nodes from two different host-wide instance graphs, these two graphs can be concatenated together. Such edges serve as the glue for concatenation, and they are usually incurred by socket-based communications: a local program communicates with a remote one via message passing, which can be captured by system call *socketcall*. Hence, the graph concatenation starts by socket identification and pairing. For example, system call "*sys_accept, start:681154, end:681162, pid:4935, pname:sshd, srcaddr:172.18.34.10, srcport:36036, sinkaddr:192.168.101.5, sinkport:22*" results in a directed edge (*172.18.34.10, 36036*)$\rightarrow$(*192.168.101.5, 22*), where a socket object instance is denoted as a tuple (*ip*, *port*).

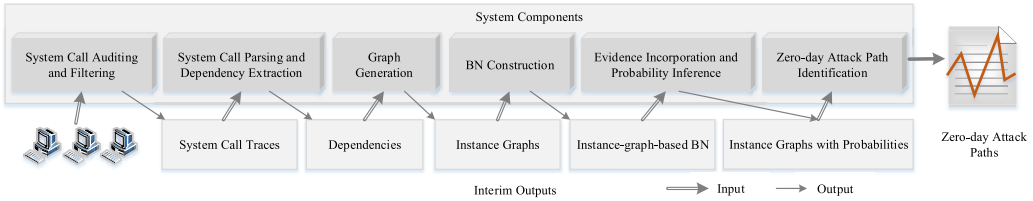


Fig. 6. System design.

This edge can be used to concatenate the host-wide instance graphs of *172.18.34.10* and *192.168.101.5*. If socket communication exists between two hosts, their instance graphs will be concatenated. Otherwise, their graphs will be isolated. A network-wide instance graph can be composed of one or more isolated graphs.

The system calls are usually collected within a specific time window and then sent to an off-line machine for analysis. In some cases, the generated instance graphs in different time windows need to be connected together. We connect these individual instance graphs with the following steps: 1) identify identical objects that appear in different windows; 2) For each of such object, connect every two closest windows by adding a state transition dependency between the last instance of the object in a window and the first instance of the same object in the next window. The last instance and first instance are determined by the timestamps of instances.

D. BN Construction

The instance graph built above provides its structure topology for BN construction in this module. The instances and dependencies in the instance graph are hence also nodes and edges in BN. This module further associates each node with a CPT table based on the settings such as ρ and τ according to the infection propagation models (detailed in Section III-A). The nodes and CPT tables are then specified in a *.net* file, which is a file type to carry the BN information. Engines such as SamIam [7] can turn a *.net* file into a graphical BN view.

E. Evidence Incorporation and Probability Inference

This module incorporates evidence into instance-graph-based BN by either directly labeling the infection state of the involved object instance as *infected*, or adding an LOM node as child node to the object instance (detailed in Section I). The CPT table associated with the LOM node is configured per the false positive and negative rates of the corresponding security sensors. After this, each node in the instance graph then receives a probability due to BN-enabled probability inference.

F. Zero-Day Attack Path Identification

The inferred probabilities guide the identification of zero-day attack paths: by preserving the nodes with high probabilities and edges interconnecting them in the instance graph, the zero-day attack paths stand out. We designed Algorithm 2, which is a DFS-based (depth-first search) algorithm, to highlight those nodes in instance graph, which either possess high probabilities on their own, or have both an ancestor and a descendant with high probabilities. The highlighted nodes are the ones that actually contribute to the infection propagation, and thus should be preserved. A tuning parameter in our system, i.e. *threshold*, is used to regulate the bottom probability that can be regarded as high probability. For example, if the threshold is set to 0.8, only instances that have the infection probabilities at 80% or higher will be preserved.

Algorithm 2 Algorithm of Zero-Day Attack Paths Identification

Require: the instance graph $G(V, E)$, a vertex $v \in V$
Ensure: the 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

V. IMPLEMENTATION

The probabilistic system designed above is implemented via a prototype named *ZePro*, through approximately 5471 lines of code, which include about 2411 lines of C code for a loadable kernel module auditing 39 system calls, and approximately 2915 lines of gawk code that generates a *.net* file for instance-graph-based BN construction and a *dot*-compatible file for visualizing the zero-day attack paths in Graphviz [8], as well as 145 lines of Java code for probability inference through the API provided by the BN engine SamIam [7].

A. System Call Auditing and OS-Aware Reconstruction

System call auditing can be achieved by hooking into the Linux system call interface, *sys_call_table*. We did this via a loadable kernel module. System calls of interest can be added or deleted for auditing, including those encapsulated in system call *socketcall*, such as *sys_accept*, *sys_recvfrom* and *sys_sendto*. In the module, each call is hooked to record parameters and return values, as well as OS-aware information from kernel data structures for object identification, such as process descriptor from *task_struct* and file descriptor from *files_struct*. In addition, timestamps are logged respectively when a system call in invoked and returned.

B. Graph Representation and Graph Conversion

We represent our graphs with an adjacency matrix (*Map*) because during the graph generation and path identification we need to quickly look up if there is already an existing edge connecting two nodes. With adjacency matrix, the query takes only O(1) time, while with other data structures it may take O($|v|$) or O($|e|$) time, where $|v|$ and $|e|$ are respectively the number of nodes and edges in a graph. For each pair of graph nodes (*src* and *sink*), there is only one edge between them as the instance graph is instance-specific.

We also support graph conversion from instance graph to SODG for result comparisons. In SODG, each pair of graph nodes (*src* and *sink*) may correspond to numerous edges, which are incurred by different system calls or the same system call at different timestamps. Our implementation aggregates them into a single one, maintaining the matrix cell (*Map*[*srcObj, sinkObj*]) to count the number of edges, and a timestamp list (*tMap*[*srcObj, sinkObj*]) to associate this aggregated edge with different timestamps.

C. Instance Graph Pruning

The introduction of the concept "instances" makes each system object may have multiple different "versions", and hence an instance graph can be very large. To reduce the complexity and speed up the processing, it's essential to perform instance graph pruning. The bottom line is not to hurt the capturing of the infection propagation process. We applied the following pruning methods towards instance graphs. In Section VI-B.2 we will discuss the impact of pruning.

First, repeated dependencies can be pruned. Between any pair of system objects it is not unusual that the same dependency may occur many times, though it may be caused by different system calls. 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 any other object since the last time that the same $src \rightarrow sink$ dependency occurred.

Second, the root instances can be pruned. Root instances refer to those nodes which have never appeared as the *sink* object in any $src \rightarrow sink$ dependency during the analysis period. For instance, *file 3* in Figure 3 only appears as the src object in the dependencies parsed from the system call trace in Figure 1a, so *file 3 instance 1* can be ignored in the simplified instance graph. No directed edges are pointing to such instances, hence they are not influenced by any other object in the analysis period, and of course they are not manipulated by attackers either. Ignoring these root instances will not break any routes of intrusion sequence and hence will not hurt the capture of infection propagation. This method is especially helpful for the situations like a process reading a large number of configuration or library header files.

Third, repeated mutual dependencies can be pruned. Two objects may keep affecting each other through creating new instances, and this will cause a lot of repeated mutual dependencies. One situation is that a process frequently sends and receives 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. Another situation is that a process keeps taking feeds from a file and then writing the output back to the file. In both situations, no other objects are involved during the two-object interaction, and hence the infection statuses of the two objects will keep the same throughout all the new instances. Based on this recognition, the instance graph can preserve only the very first and last dependencies while neglect the intermediate ones.

VI. EXPERIMENTS

A. Experimental Setup

For evaluation, we built a web-shop test-bed to emulate a small-scale real-world enterprise network. To compare the experiment results with Patrol, we purposively used the same experiment setup as Patrol. We launched three-step attacks towards it under the surveillance of some existing popular security sensors (such as firewalls, Snort [5], Tripwire [6], Wireshark [9], and Ntop [10]) and our system. The sensors are deployed to detect and alert known attacks, which can then be leveraged as intrusion evidence for BN analysis. The test-bed hosts are typically deployed with Dell PowerEdge T310 with two 2.53GHz Intel(R) Xeon(R) X3440 quad-core processor and 4GB of RAM running 32-bit Linux 2.6.24 through 2.6.32.

Figure 7 illustrates the attack scenario. Step 1, the attacker exploits vulnerability CVE-2008-0166 [11] 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

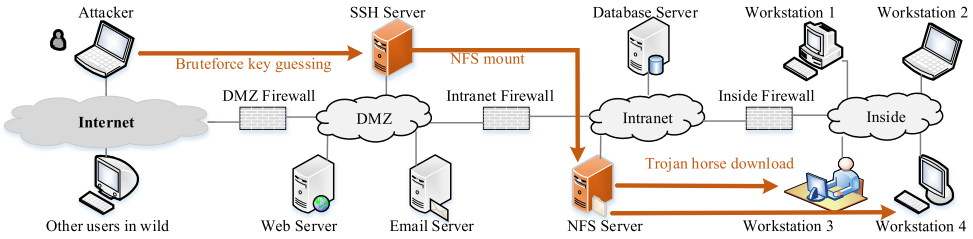


Fig. 7. Attack scenario.

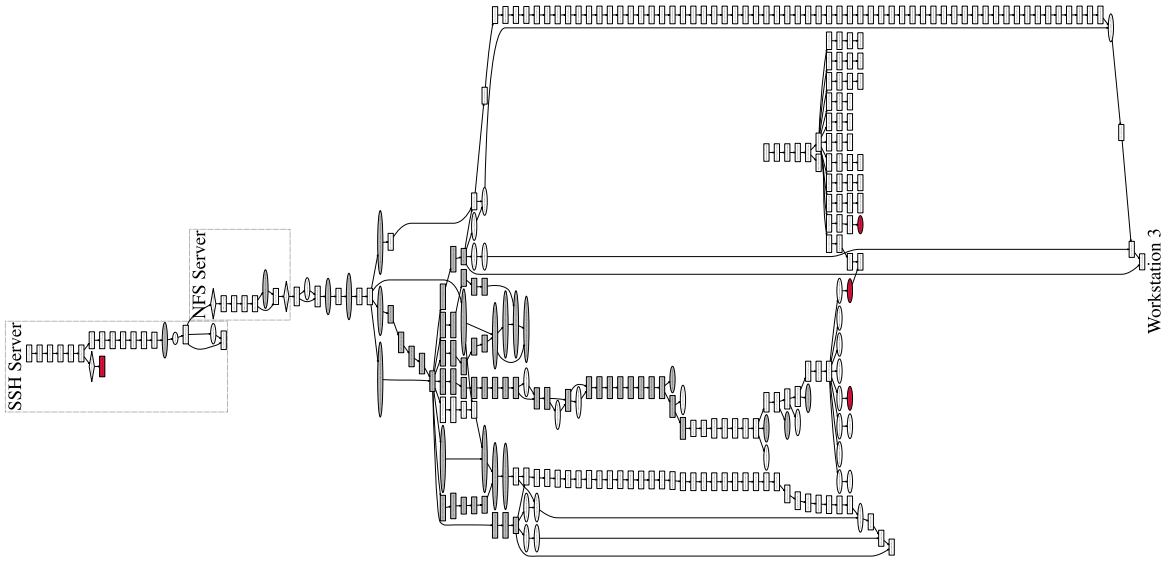


Fig. 8. The zero-day attack path in the form of an instance graph for experiment 1.

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. In one experiment, the malicious file contains a Trojan-horse that exploits CVE-2009-2692 [12] existing in the Linux kernel of workstation 3. CVE-2009-2692 is a vulnerability that allows local users to gain privileges by triggering a NULL pointer dereference. In the other experiment, the Trojan-horse exploits CVE-2011-4089 [13] that lies in bzexe command on workstation 4. CVE-2011-4089 allows local users to execute arbitrary code by pre-creating a temporary directory.

Since it's usually hard to acquire zero-day vulnerabilities, we emulate them with known ones. For example, CVE-2009-2692 is treated as a zero-day vulnerability by assuming the current time is Dec 31, 2008. Moreover, other security holes such as configuration errors on NFS can also be viewed as a special type of unknown vulnerability as long as they are ruled out by vulnerability scanners like Nessus. An extra benefit of emulation is that the information for these known zero-day vulnerabilities can be available to verify the correctness of our experiment results.

B. Experiment Results

While constantly collecting system calls and security alerts, we conducted the three-step attacks towards the testbed. In experiment 1, we collected 143120 system calls generated by three hosts in 40 minutes, and constructed an instance-graph-based BN with 1853 nodes and 2249 edges. Since experiment 2 just differs from experiment 1 in attack step 3, in experiment 2 we only analyzed 54998 system calls generated by workstation 4. The constructed BN contains 911 nodes and 1214 edges. The evidence as in Table I is collected and fed into the two BNs respectively.

TABLE I
THE COLLECTED EVIDENCE

Exp	ID	Host	Evidence
Exp 1	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"
Exp 2	E5	Workstation 4	Tripwire reports "/symlinkattack.o is added"
	E6	Workstation 4	Tripwire reports "/virus is added"

Figure 8 and Figure 9 show the identified zero-day attack paths in the form of instance graphs for experiment 1 and 2. The graph uses rectangles, ellipses, and diamonds to represent processes, files, and sockets respectively. The parameter settings in this experiment is: the intrinsic infection rate ρ is set as 0.0001; the contact infection rates τ is 0.9; and the probability threshold of recognizing high-probability nodes is 80%. The evidence fed into BN are highlighted with red color, and nodes verified to be malicious are marked with grey color.

1) Correctness: The experiment results demonstrate that ZePro is able to reconstruct the attack story-line, such as how the attack has happened, which files have been touched, and whether the raised alerts have correlations, etc. For example,

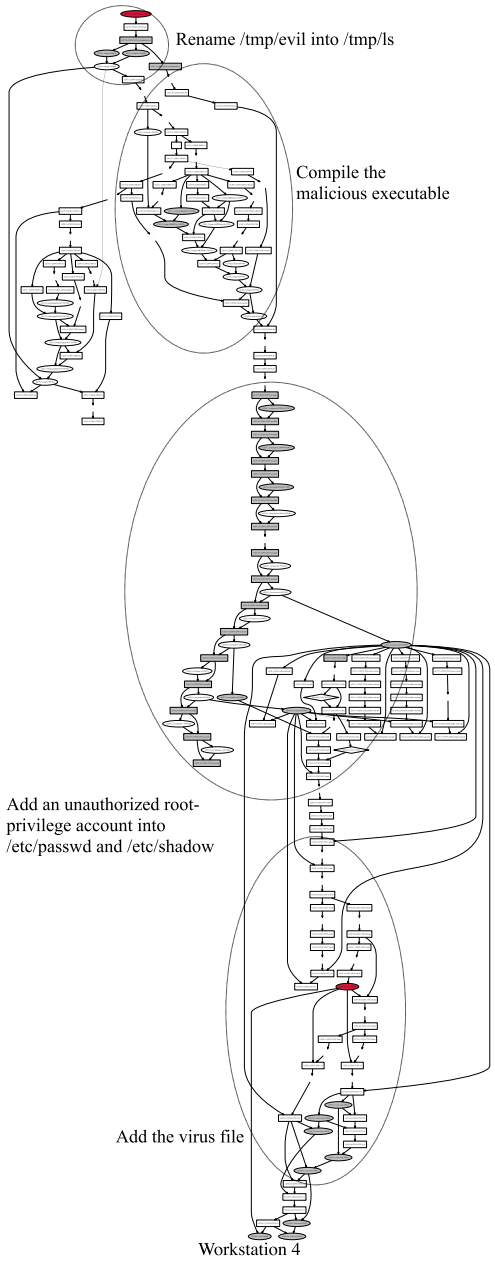


Fig. 9. The zero-day attack path in the form of an instance graph for experiment 2.

by scrutinizing Figure 8, we noticed that nodes with high infection probabilities are all related to attackers' malicious activities along the road. Figure 8 successfully captures the infection propagation process and shows how the trojan horse is uploaded from SSH server to NSF server, and then gets executed on workstation 3. Similarly, Figure 9 captures the process of renaming */tmp/evil* into */tmp/ls*, and leveraging */tmp/ls* for further malicious activities such as adding an unauthorized root-privilege account into */etc/passwd* and */etc/shadow*.

A merit of our approach is that the identified path provides a context to reveal the hidden exploits, even when no evidence is provided for the particular exploits. For example, in experiment 1, there is no intrusion evidence provided by security sensors for NFS server. Nevertheless, the zero-day attack path in Figure 8 can still capture how NFS server contributes to the entire infection propagation process: the malcious file *workstation_attack.tar.gz* is uploaded from SSH Server to the shared directory */exports* on NFS Server, and then downloaded to */mnt* on workstation 3.

Furthermore, the identified path enables the recognition of zero-day exploits by exposing related critical objects. For instance, the appearance of */exports* object on the path indicates possible configuration errors of NFS server 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 also has high infection probability on the path. A further check on this page-zero object reveals an exploit: the object triggers the null pointer dereference and enables attackers gain privilege on workstation 3. Therefore, exposing critical attack-related system objects on attack paths can help security admins catch potential zero-day exploits.

In addition, the instance-graph-based BN can capture state transitions of an object using instances. This greatly helps with the understanding of the infection propagation process. Since an instance is a "version" of an object at a specific time, the change of infection probabilities for instances reflects the change of states for the same object. By matching the instances and dependencies to the system call traces, we can even find out the system call that causes the state-changing of the object. For instance, in Figure 8, the node $x2086.4$*:(6763:6719:tar)* represents an instance of process (*pid:6763, pcmd:tar*) at a specific time t. Before t, the process was regarded as innocent because previous instances have low infection probabilities. The process becomes highly suspicious after a dependency is added between node $x2082.2$*:(/home/user/testbed/workstation_attack.tar.gz:1384576)* and node $x2086.4$. Mapping the dependency back to the system call traces proves that the state-changing of the process is caused by a system call "*syscall:read, start:827189, end:827230, pid:6763, ppid:6719, pcmd:tar, ftype:REG, pathname:/home/user/testbed/workstation_attack.tar.gz, inode:1384576*", which indicates that the process reads a suspicious file.

2) Size of Instance Graph and Zero-Day Attack Paths: One concern of adopting instance graph is that it can become too large due to introduction of instances. However, the techniques of pruning instance graphs can significantly reduce the number of instances. Table II shows the pruning results for experiment 1. The pruning is very effective in reducing the size of instance graphs. Take SSH Server for example, the number of objects involved in the system call log is 349. Without pruning, the generated instance graph contains 10447 instance. After pruning, the number of instances is reduced to 745. In total, the number of instances in the network-wide instance graph for experiment 1 decreases from 39840 to 1853. On average the object to instance ration is 1:2.03, which is very reasonable. In addition, by merging instances belonging to the same object into one node, ZePro can convert the form of zero-day attack paths from instance graphs to SODGs. SODG based zero-day attack paths is composed of only objects, and can be used for analysis when instance information is not necessary. Figure 10 and Figure 11 are respectively the SODG form of zero-day attack paths for Figure 8 and Figure 9.

Compared with the results in [1], Figure 8 and Figure 9 show that ZePro substantially outperforms Patrol. Even when Patrol is provided with extensive pre-knowledge for common features of known exploitations at OS-level, ZePro generates

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

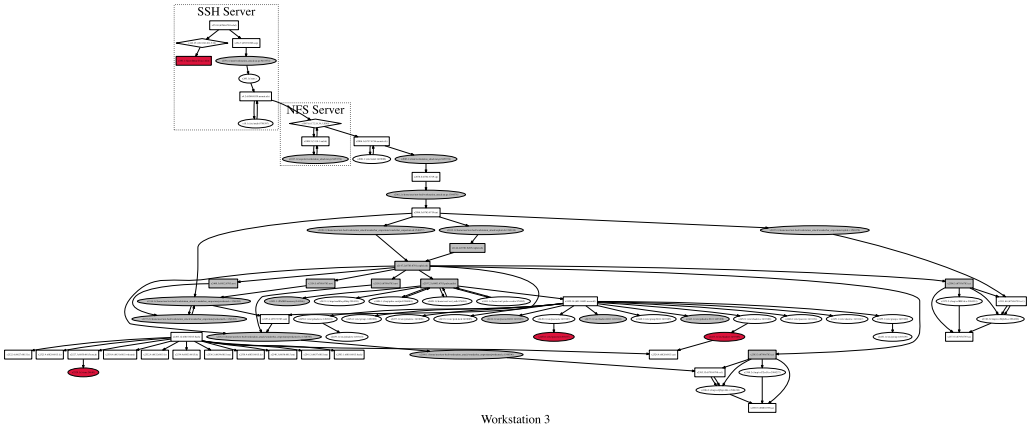


Fig. 10. The object-level zero-day attack path for experiment 1.

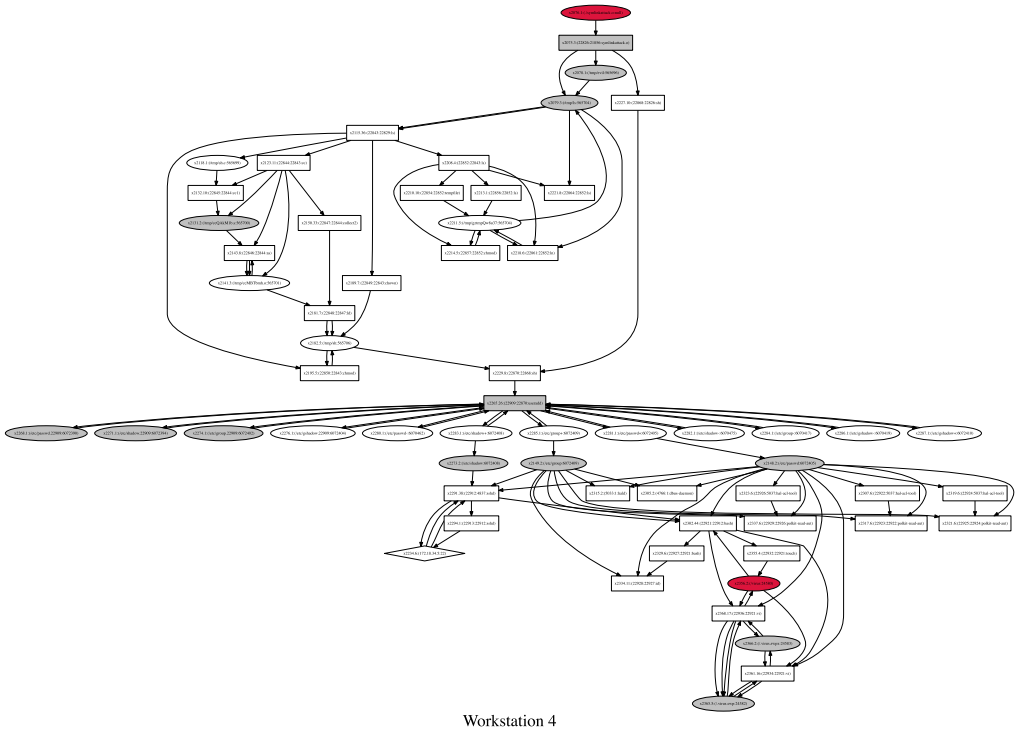


Fig. 11. The object-level zero-day attack path for experiment 2.

equal or much better results than Patrol. Specifically, for experiment 2, the size of zero-day attack paths revealed by Patrol and ZePro are very close. The path by Patrol has 60 nodes and the path by ZePro has 61 nodes (Figure 11). This is because the system call log analyzed in this experiment is relatively small, the objects identified in these paths are

TABLE III
THE INFLUENCE OF EVIDENCE OF EXPERIMENT 1

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 EVIDENCE IN EXPERIMENT 2

Evidence	***Workstation 4***				
	x2078.1	x2079.3	x2265.26	x2273.2	x2148.2
No Evi.	0.05%	0.75%	1.51%	0.93%	0.91%
E5	64.01%	74.43%	54.63%	34.95%	34.94%
E6	79.82%	93.63%	98.82%	65.63%	68.82%

already the smallest set of suspicious objects to constitute the paths. When a larger set of system calls is analyzed, ZePro generates candidate paths much smaller than Patrol without hurting the correctness of paths. For example, in experiment 1, the zero-day attack path identified by Patrol has 175 objects, while the path by ZePro contains 77 objects (Figure 10). Given that only 913 objects are involved in the original instance graph before path identification, 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 and generates same results, but Patrol is not able to discern true zero-day attack paths from suspicious ones. For instance, 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.

C. Evaluation and Discussion

1) Influence of Evidence: A number of nodes in Figure 8 are selected as the representative instances of interest. Table III and Table IV show how the infection probabilities of these instances vary with each piece of evidence fed into BN. We assume the evidence is observed in the order of attack sequence.[1] In Table III, 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. In Table IV, similar probability updates are observed with the appearence of *E5* adn *E6*. Hence, the evidences makes the zero-day attack path's instances emerge gradually from the "sea" of nodes in the instance graph. It is also possible that the arrival of new evidence may decrease probabilities of certain instances, which will then get removed from the ultimate path. In a nutshell, the more evidence fed into BN, the better results that get closer to the ground truth.

2) Influence of False Alerts: We also want to know whether the false alerts would mislead the probability inference and cause the final identification results incorrect. We use the Tripwire alert *E4* for example by assuming it as a false alarm and observe its impact to the BN output. According to Table V, when only one more piece of evidence is fed into BN, *E4* will greatly influence the probabilities of some instances on Workstation 3; however, when more evidence comes up, the influence of *E4* drops. For instance, given *E1* as the only extra evidence, the infection probability of $x2006.2$ is 97.78% when *E4* is true, and 29.96% when *E4* is false. Nonetheless, if all other evidence is also fed into BN, the infection probability of $x2006.2$ only goes from 81.13% to 81.3% if *E4* turns out false. Overall, the influence of false alerts could be counteracted by more true alerts.

3) Influence of τ and ρ: Sensitivity analysis is also performed to evaluate the influence of the contact infection rate τ and the intrinsic infection rate ρ through tuning them. ρ is usually set as a very small value. The experiment results are not very sensitive to the value of ρ. However, τ turns out to be influential for the probabilities produced by BN, since it decides how likely $sink_j$ get infected given src_i is already infected via a $src_i \rightarrow sink_j$ dependency. If a node is labeled as "infected", the nodes that are directly or indirectly connected to this node will tend to have higher infection probabilities when τ gets bigger. To evaluate the impact of τ, we tuned τ to values between 0 and 1 with 0.1 as a step, and then analyzed the probabilities generated through BN inference. Our experiments show that a small adjustment of τ (e.g. changing it from 0.9 to 0.8) does not much influence the output probabilities, but a major adjustment of τ (e.g. changing it from 0.9 to 0.5) can largely affect the probabilities. Despite of such influence of τ towards the produced infection probabilities, it will not greatly affect the identification of zero-day attack paths, as the probability threshold of recognizing relatively higher-probability nodes for zero-day attack paths can be adjusted according to the value of τ. For example, when τ is small (like 50%), the threshold could be also lowered to reveal those nodes with relatively higher infection probabilities. Even probabilities around 40%-60% should be considered as highly suspicious because it is hard for an instance to get infected with such a low contact infection rate.

4) Complexity and Scalability: The performance evaluation for ZePro are conducted towards both online system call logging and off-line data analysis. By measuring with UnixBench and kernel compilation, the run-time performance overhead caused by the system call logging component is around 15% to 20% for the system. The time cost for off-line data analysis includes three major parts: instance-graph-based BN generation, BN probability inference and zero-day attack path identification. Since the DFS algorithm is applied towards every node in the instance graph, the time

[1]Given the same set of evidence, the evidence input order does not affect the final probability inference results.

TABLE V
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%

complexity for both BN generation and path identification is $O(|V|^2)$. The time cost for probability inference is determined by the algorithm used in SamIam. In our experiments, the off-line analysis is conducted on a machine with 2.4 GHz Intel Core 2 Duo processor and 4G RAM. Table II summarizes the time cost of BN generation for each host. Therefore, the total time spent on BN construction is around 27 seconds. For probability inference, the algorithm used in our experiments is *recursive conditioning*. For a BN with 1854 nodes, the average time cost is 1.57 seconds for BN compilation and probability inference, and 59 seconds for zero-day attack path identification. By adding all the time costs together, the average off-line data analysis speed is 280 KB/s. In addition, the average memory used for BN compilation is 4.32 Mb.

The scalability of our approach is ensured by several aspects. 1) The time window of system call analysis is adjustable. For instance, the system calls can be collected and sent to an off-line machine for analysis every 30 or 40 minutes. In our experiments, a 40-minute system call log generates a BN with 1854 nodes. Although the BN size does not have a deterministic relation with the number of system call analyzed, a smaller time window usually generates a smaller BN. 2) In a large network with many hosts, the instance graphs for each individual hosts may not necessarily connect to each other. A network-wide instance graph is thus composed of one or more isolated individual graphs. In this case, the size of individual BNs won't become too large. 3) The instance graph generation and zero-day attack path identification can be conducted in parallel. To make an estimation based on our experiment results, we can consider an enterprise network with 10000 hosts. If the off-line analysis is conducted on a cluster with 512 processors, the time used for instance graph generation and zero-day attack path identification is respectively around 2.93 minutes and 6.3 minutes. 4) The scalability of BN compilation and probability inference has been intensively studied. Interested readers may refer to [14] and [15] for more information. For example, the recursive conditioning [16] algorithm used in this paper provides scalable BN inference by supporting a smooth tradeoff between time and space.

VII. RELATED WORK

In this section, we compare ZePro with other research works by examining target problems, methodologies, and employed techniques, etc.

A. Zero-Day Attack Detection

This paper is related to zero-day attack detection due to its nature. It is well known that signature-based intrusion detection systems [5], [73] could not cope with zero-day attacks very well. Instead, anomaly detection [74]–[88] and specification-based detection [89]–[91] can help detect zero-day exploits. Anomaly detection, including system call-based ones [74], [76], [83], [84], profiles the normal behavior of programs or systems and is alerted to any deviations. However, anomaly detection suffers from large false positives. Specification-based detection improves accuracy based on application-specific policies, but requires experts' knowledge and experience to specify them. This process is very time-consuming and error-prone. Considering the extreme difficulty of detecting individual zero-day attacks, our paper shows that identifying an attack path containing the zero-day exploits is a substantially more feasible strategy.

B. Provenance Tracking

Provenance tracking may result in the discovery of attack paths. A number of provenance tracking systems have been proposed to monitor and parse system activities, mostly for the purpose of intrusion forensics or recovery. One of the pioneering work is Backtracker [3], which performs back tracking through system object dependencies to identify intrusion provenance. This philosophy was followed by [1], [4], [18]–[31]. References [1], [4], [18]–[27] track information flows at system call level, and [28]–[31] track at finer granularities at the cost of higher overheads. Among the work, BEEP [30], ProTrace [26] and MPI [31] seek to gain more precision than Backtracker [3] via execution partition, but may have limited scalability due to not-always automated instrumentation. SLEUTH [27] makes efforts for more efficient event storage and analysis (and thus better scalability). It leverages tags to achieve more precision in attack detection and root-cause identification than Backtracker.

Different from all the above work, this paper pursues the identification of paths *across multiple interconnected machines* to reveal zero-day exploits, *with no tagging or any other prior information*. The most related work is our predecessor work Patrol [1]. This work targets the same problem with Patrol, but is substantially different in several aspects: 1) Patrol relies on extensive pre-knowledge regarding known vulnerability exploitations to distinguish zero-day attack paths from the huge number of candidate paths, while our approach does not require any pre-knowledge, and solely rely on collected intrusion evidences; 2) Patrol only conducts qualitative analysis and treats every object on the identified paths as having the same malicious status, while our approach quantifies the infection status of each system object with probabilities; 3) Patrol performs dependency tracking to generate a huge candidate pool for zero-day attack paths, while our system relies on the computed probabilities to reveal higher probability objects as suspicious ones.

C. Event Causality/Dependency Discovery

Event causality or dependency discovery can also lead to the formation of causal graphs, which potentially reveal abnormal paths. This can be done based on workflow-centric distributed system monitoring and tracing, including both the industry platforms [33]–[37] and the academic ones [38]–[58]. Some of the solutions are network-based methods [38]–[47], and others are host-based [48]–[58]. Compared with network-based solutions, host-based solutions gain better accuracy

and faster analysis speed at the cost of performance degradation. The degradation is caused by required extra instrumentation on traced systems. Specifically, host-based solutions instrument to propagate metadata across distributed systems [33], [50]–[54], [57], or require programmers to write temporal join-schemas to introduce causal relationships among variables [48], [49]. In contrast, network-based solutions correlate variables or timings for causal inference [32].

This paper is a host-based solution. It differs by conducting instrumentations and probabilistic inference at the system call level, which is more fine-grained than the above methods.

D. Bayesian Networks

Bayesian network has been used in a number of research problems, such intrusion detection [96]–[99], security analysis [95], security measurement and risk management [100]–[102]. For example, Kruegel et al. used Bayesian network to aggregate outputs provided by different anomaly-based intrusion detection systems. Xie *et al.* [95] built an example Bayesian network based on the dependency attack graph to perform real-time security analysis. Poolsappasit *et al.* [102] developed a risk management framework based on Bayesian networks to quantify the chances of network compromise. However, none of the above methods studied the problem of zero-day attack path detection. Moreover, none of them construct large-scale Bayesian networks at the operating system level.

E. Information Flow Capture

Information flow can be a natural reflection of workflow. That is, controlling the information flow can help enforce policies and prevent anomalies, and capturing the information flow can help extract abnormal paths. Several works have studied on how to specify, control and mandate information flow in systems [59]–[63]. Panorama [64] taints suspicious objects, leverages the information flow capture to detect malware, and gets a taint graph (inherently attack path) visualizing the exploit. However, such information flow capture still heavily relies on the specification of sources to taint, and doesn't go beyond individual machines yet. In contrast, our paper is an effort to capture cross-machine information flow with no tainting.

F. Alert Correlation

Attack paths can also be resulted from alert correlation, which is a technique to aggregate similar alerts, prioritize important alerts, and connect apart alerts through causal correlation [27], [65]–[72]. Among the existing solutions, [27], [70], [71] start to leverage system call-based provenance tracking to help causally correlate scattered alerts and reconstruct attack scenarios. Others rely on expert knowledge and experience about the relationships between the alert types. These approaches end with the reconstruction of attack scenarios based on alerts or policies. Our paper makes nontrivial efforts to further pursue zero-day attack path detection, by bringing in the Bayesian Networks that take alerts as evidence and quantitatively compute the infection probabilities of system objects. This leads to discovery of previously-unknown infection objects other than the alerts themselves.

G. Attack Graphs

Attack graphs can generate possible attack paths by analyzing the vulnerabilities existing in a network. There are mainly two types of attack graphs: state enumeration attack graph [103]–[108] (also called network-state attack graph [110]) and dependency attack graph [109]–[116]. The major differences between our approach and the attack graph are: 1) Attack graphs only deal with known vulnerabilities. Since usually zero-day vulnerabilities cannot be scanned out by vulnerability scanners, attack graph is not able to generate attack paths containing zero-day vulnerabilities. 2) Our object instance graph captures system activities that is happening in real time, but an attack graph shows possible exploitation sequences that might happen. 3) The two type of graphs are at different abstract levels. The object instance graph is at the low system object level and captures the fine-grained system activities, while the attack graph is at the higher application level.

Attack graphs have also been employed to measure the security risks caused by zero-day attacks [93], [94]. The metric simply counts the number of unknown vulnerabilities required to compromise an asset, rather than detects the actual zero-day exploits. This is different from our work.

VIII. Limitation and Conclusion

The system still needs to address some limitations in furture work. For example, when some attack activities evade system calls (it's difficult, but possible), or the attack time span exceeds the analysis time period, the constructed instance graphs may not capture the complete zero-day attack paths. In such cases, our system can only reveal parts of the paths.

Overall, this paper proposes to use Bayesian networks to identify the zero-day attack paths. For this purpose, an object-graph-based BN is defined and constructed. By incorporating the intrusion evidence and computing the probabilities of objects being infected, the system can successfully reveal the zero-day attack paths.

References

[1] J. Dai, X. Sun, and P. Liu, "Patrol: Revealing zero-day attack paths through network-wide system object dependencies," in *Proc. ESORICS*, 2013, pp. 536–555.
[2] X. Sun, J. Dai, P. Liu, A. Singhal, and J. Yen, "Towards probabilistic identification of zero-day attack paths," in *Proc. CNS*, Oct. 2016, pp. 64–72.
[3] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 223–236, 2003.
[4] X. Xiong, X. Jia, and P. Liu, "Shelf: Preserving business continuity and availability in an intrusion recovery system," in *Proc. ACSAC*, Dec. 2009, pp. 484–493.
[5] *Snort*. Accessed: Mar. 1, 2018. [Online]. Available: https://www.snort.org/
[6] *Tripwire*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.tripwire.com/
[7] *SamIam*. Accessed: Mar. 1, 2018. [Online]. Available: http://reasoning.cs.ucla.edu/samiam/
[8] *GraphViz*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.graphviz.org/
[9] *Wireshark*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.wireshark.org
[10] *Ntop*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.ntop.org
[11] *CVE-2008-0166*. Accessed: Mar. 1, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166
[12] *CVE-2009-2692*. Accessed: Mar. 1, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692
[13] *CVE-2011-4089*. Accessed: Mar. 1, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4089
[14] O. J. Mengshoel, "Understanding the scalability of Bayesian network inference using clique tree growth curves," *Artif. Intell.*, vol. 174, pp. 984–1006, Aug. 2010.

[15] V. K. Namasivayam and V. K. Prasanna, "Scalable parallel implementation of exact inference in Bayesian networks," in *Proc. ICPADS*, Jul. 2006, pp. 1–8.

[16] A. Darwiche, "Recursive conditioning," *Artif. Intell.*, vol. 126, pp. 5–41, Feb. 2001.

[17] L. Bilge and T. Dumitras, "Before we knew it: An empirical study of zero-day attacks in the real world," in *Proc. ACM CCS*, 2012, pp. 833–844.

[18] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 163–176, 2001.

[19] A. Goel, W.-C. Feng, D. Maier, and J. Walpole, "ForenSix: A robust, high-performance reconstruction system," in *Proc. ICDCS*, Jun. 2005, pp. 155–162.

[20] K. M.-Reddy., D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *Proc. USENIX ATC*, 2006, pp. 43–56.

[21] U. Braun, S. Garfinkel, D. A. Holland, K.-K. M.-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *Proc. Int. Provenance Annotation Workshop*, 2006, pp. 171–183.

[22] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in *Proc. Int. Middleware Conf.*, 2012, pp. 101–120.

[23] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting high-fidelity whole-system provenance," in *Proc. ACSAC*, 2012, pp. 259–268.

[24] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proc. USENIX Secur.*, 2015, pp. 319–334.

[25] X. Kiang, A. Walters, D. Xu, E. H. Spafford, Y.-M. Wang, and F. Buchholz, "Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach," in *Proc. ICDCS*, Jul. 2006, p. 38.

[26] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proc. NDSS*, 2016, pp. 1–15.

[27] M. N. Hossain *et al.*, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *Proc. USENIX Secur.*, 2017, pp. 487–504.

[28] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 121–132, 2012.

[29] S. Arzt *et al.*, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[30] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Proc. NDSS*, 2013, pp. 1–16.

[31] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantics aware execution partitioning," in *Proc. USENIX Secur.*, 2017, pp. 1111–1128.

[32] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? Key design insights from years of practical experience," Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14, 2014.

[33] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc, Mountain View, CA, USA, Tech. Rep. dapper-2010-1, Apr. 2010.

[34] *AppNeta TraceView*. Accessed: Mar. 1, 2018. [Online]. Available: http://appneta.com

[35] *Compuware*. Accessed: Mar. 1, 2018. [Online]. Available: http://www.compuware.com

[36] *Apache HTrace*. Accessed: Mar. 1, 2018. [Online]. Available: http://htrace.incubator.apache.org/

[37] *Zipkin, a Distributed Tracing System*. Accessed: Mar. 1, 2018. [Online]. Available: https://zipkin.io/

[38] A. Keller, U. Blumenthal, and G. Kar, "Classification and computation of dependencies for distributed management," in *Proc. ISCC*, 2000, pp. 78–83.

[39] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 74–89, 2003.

[40] P. V. Bahl *et al.*, "Discovering dependencies for network management," in *Proc. HotNets*, 2006, pp. 1–6.

[41] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat, "WAP5: Black-box performance debugging for wide-area systems," in *Proc. WWW*, 2006, pp. 347–356.

[42] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 13–24, 2007.

[43] D. Dechouniotis, X. Dimitropoulos, A. Kind, and S. Denazis, "Dependency detection using a fuzzy engine," in *Proc. Int. Workshop Distrib. Syst., Oper. Manage.*, 2007.

[44] S. Kandula, R. Chandra, and D. Katabi, "What's going on?: Learning communication rules in edge networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 87–98, 2008.

[45] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions," in *Proc. USENIX OSDI*, 2008, pp. 117–130.

[46] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson, "NSDMiner: Automated discovery of Network Service Dependencies," in *Proc. INFOCOM*, Mar. 2012, pp. 2507–2515.

[47] B. Peddycord, III, P. Ning, and S. Jajodia, "On the accurate identification of network service dependencies in distributed systems," in *Proc. LISA*, 2012, pp. 181–194.

[48] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online Modelling and Performance-aware Systems," in *Proc. HotOS*, 2003, pp. 85–90.

[49] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *Proc. OSDI*, 2004, p. 18.

[50] Y. M. Chen *et al.*, "Path-based failure and evolution management," in *Proc. NSDI*, 2004, pp. 1–14.

[51] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. NSDI*, 2006, p. 9.

[52] E. Thereska *et al.*, "Stardust: Tracking activity in a distributed storage system," in *Proc. ACM SIGMETRICS*, 2006, pp. 3–14.

[53] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *Proc. NSDI*, 2007, pp. 20.

[54] A. Chanda, L. Alan Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," in *Proc. EuroSys*, 2007, pp. 1–14.

[55] P. Barham *et al.*, "Constellation: Automated discovery of service and host dependencies in networked systems," Microsoft Res., Washington, DC, USA, Tech. Rep. MSR-TR-2008-67, 2008.

[56] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft, "Macroscope: End-point approach to networked application dependency discovery," in *Proc. ACM CoNEXT*, 2009, pp. 229–240.

[57] R. R. Sambasivan *et al.*, "Diagnosing performance changes by comparing request flows," in *Proc. NSDI*, 2011, p. 1.

[58] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proc. SOSP*, 2015, pp. 378–393.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. OSDI*, 2006, pp. 263–278.

[60] M. Krohn *et al.*, "Information flow control for standard OS abstractions," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 321–334, 2007.

[61] Z. Mao, N. Li, H. Chen, and X. Jiang, "Combining discretionary policy with mandatory information flow in operating systems," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 3, p. 24, 2011.

[62] W. K. Sze, B. Mital, and R. Sekar, "Towards more usable information flow policies for contemporary operating systems," in *Proc. SACMAT*, 2014, pp. 75–84.

[63] W. K. Sze and R. Sekar, "Provenance-based integrity protection for windows," in *Proc. ACSAC*, 2015, pp. 211–220.

[64] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. ACM CCS*, 2007, pp. 116–127.

[65] H. Debar and A. Wespi, "Aggregation and correlation of intrusion-detection alerts," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2001.

[66] A. Valdes and K. Skinner, "Probabilistic alert correlation," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2001.

[67] X. Qin and W. Lee, "Statistical causality analysis of infosec alert data," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2003.

[68] P. Ning and D. Xu, "Learning attack strategies from intrusion alerts," in *Proc. ACM CCS*, 2003, pp. 200–209.

[69] S. Noel, E. Robertson, and S. Jajodia, "Correlating intrusion events and building attack scenarios through attack graph distances," in *Proc. ACSAC*, 2004, pp. 350–359.

[70] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *Proc. NDSS*, 2005, pp. 1–12.

[71] Y. Zhai, P. Ning, and J. Xu, "Integrating IDS alert correlation and OS-Level dependency tracking," in *Proc. Int. Conf. Intell. Secur. Inform.*, 2006.
[72] W. Wang and T. E. Daniels, "A graph based approach toward network forensics analysis," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 1, p. 4, 2008.
[73] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, nos. 23–24, pp. 2435–2463, 1999.
[74] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE SP*, May 1996, pp. 120–128.
[75] A. P. Kosoresow and S. A. Hofmeyer, "Intrusion detection via system call traces," *IEEE Software*, vol. 14, no. 5, pp. 35–42, Sep. 1997.
[76] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, 1998.
[77] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proc. IEEE SP*, May 1999, pp. 133–145.
[78] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Proc. IEEE SP*, May 1999, pp. 120–132.
[79] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE SP*, May 2000, pp. 144–155.
[80] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. IEEE SP*, May 2000, pp. 156–168.
[81] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE SP*, May 2003, pp. 62–75.
[82] C. Kruegel and G. Vigna, "Anomaly detection of Web-based attacks," in *Proc. ACM CCS*, 2003, pp. 251–261.
[83] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Proc. ESORICS*, 2003, pp. 326–343.
[84] G. Tandon and P. K. Chan, "Learning rules from system call arguments and sequences for anomaly detection," in *Proc. ICDM DMSEC*, 2003, pp. 1–13.
[85] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proc. ACM CCS*, 2004, pp. 318–329.
[86] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. IEEE SP*, May 2006, pp. 1–15.
[87] K. Berlin, D. Slater, and J. Saxe, "Malicious behavior detection using windows audit logs," in *Proc. Artif. Intell. Secur.* 2015, pp. 35–44.
[88] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proc. ACM CCS*, 2015, pp. 401–413.
[89] C. Ko, M. Ruschitzka, and K. Levitt, "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," in *Proc. IEEE SP*, May 1997, pp. 175–187.
[90] P. Uppuluri and R. Sekar, "Experiences with specification based intrusion detection," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2001.
[91] R. Sekar *et al.*, "Specification-based anomaly detection: A new approach for detecting network intrusions," in *Proc. ACM CCS*, 2002, pp. 265–274.
[92] C. Kruegel, F. Valeur, and G.Vigna, *Intrusion Detection and Correlation: Challenges and Solutions*. New York, NY, USA: Springer, 2005.
[93] L. Wang, S. Jajodia, A. Singhal, and S. Noel, "*k*-zero day safety: Measuring the security risk of networks against unknown attacks," in *Proc. ESORICS*, 2010, pp. 573–587.
[94] M. Albanese, S. Jajodia, A. Singhal, and L. Wang, "An efficient approach to assessing the risk of zero-day vulnerabilities," in *Proc. SECRYPT*, Jul. 2013, pp. 1–12.
[95] P. Xie, J. H. Li, X. Ou, P. Liu, and R. Levy, "Using Bayesian networks for cyber security analysis," in *Proc. DSN*, Jun. 2010, pp. 211–220.
[96] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur, "Bayesian event classification for intrusion detection," in *Proc. ACSAC*, Dec. 2003, pp. 14–23.
[97] A. A. Sebyala, T. Olukemi, L. Sacks, and L. Sacks, "Active platform security through intrusion detection using naive Bayesian network for anomaly detection," in *Proc. London Commun. Symp.*, 2002, pp. 1–5.
[98] W. Tylman, "Anomaly-based intrusion detection using Bayesian networks," in *Proc. DepCos-RELCOMEX*, Jun. 2008, pp. 211–218.
[99] F. Jemili, M. Zaghdoud, and M. B. Ahmed, "A framework for an adaptive intrusion detection system using Bayesian network," in *Proc. IEEE Intell. Secur. Inform.*, May 2007, pp. 66–70.
[100] M. Frigault, L. Wang, S. Jajodia, and A. Singhal, "Measuring the overall network security by combining CVSS scores based on attack graphs and bayesian networks," in *Network Security Metrics*. Cham, Switzerland: Springer, 2017.
[101] M. Frigault, L. Wang, A. Singhal, and S. Jajodia, "Measuring network security using dynamic Bayesian network," in *Proc. ACM Workshop Quality Protection*, 2008, pp. 23–30.
[102] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using Bayesian attack graphs," *IEEE Trans. Depend. Sec. Comput.*, vol. 9, no. 1, pp. 61–74, Jan. 2012.
[103] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proc. IEEE SP*, May 2002, pp. 273–284.
[104] O. Sheyner and J. Wing, "Tools for generating and analyzing attack graphs," in *Proc. Int. Symp. Formal Methods Compon. Objects*, 2003.
[105] C. R. Ramakrishnan and R. Sekar, "Model-based analysis of configuration vulnerabilities," *J. Comput. Secur.*, vol. 10, pp. 189–209, Jan. 2002.
[106] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," in *Proc. IEEE Comput. Secur. Found. Workshop*, Jun. 2002 pp. 49–63.
[107] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proc. Workshop New Secur. Paradigms*, 1998, pp. 71–79.
[108] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-attack graph generation tool," in *Proc. DARPA Inf. Survivability Conf. Expo. II*, 2001, pp. 307–321.
[109] S. Noel and S. Jajodia, "Managing attack graph complexity through visual hierarchical aggregation," in *Proc. ACM Workshop Vis. Data Mining Comput. Secur.*, 2004, pp. 109–118.
[110] S. Jajodia, S. Noel, and B. OBerry, "Topological analysis of network attack vulnerability," in *Managing Cyber Threats*. Boston, MA, USA: Springer, 2005.
[111] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proc. ACM CCS*, 2002, pp. 217–224.
[112] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *Proc. ACSAC*, Dec. 2006, pp. 121–130.
[113] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs, "Efficient minimum-cost network hardening via exploit dependency graphs," in *Proc. ACSAC*, Dec. 2003, pp. 86–95.
[114] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proc. ACM CCS*, 2006, pp. 336–345.
[115] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A logic-based network security analyzer," in *Proc. USENIX Secur.*, 2005, p. 8.
[116] H. Huang, S. Zhang, X. Ou, A. Prakash, and K. Sakallah, "Distilling critical attack graph surface iteratively through minimum-cost sat solving," in *Proc. ACSAC*, 2011, pp. 31–40.

Xiaoyan Sun received the Ph.D. degree in information sciences and technology from Penn State University in 2016. She is currently an Assistant Professor with the Department of Computer Science, California State University, Sacramento, CA, USA. Her research interests include computer and network security.

Jun Dai received the Ph.D. degree in information sciences and technology from Penn State University in 2014. He is currently an Assistant Professor with the Department of Computer Science, California State University, Sacramento, CA, USA, where he is also the Director of the Center for Information Assurance and Security. His research interests include computer and network security.

Peng Liu received the Ph.D. degree from George Mason University in 1999. He is a Full Professor of information sciences and technology, and founding Director of the Center for Cyber-Security, Information Privacy, and Trust, Penn State University. His research interests include computer and network security.

Anoop Singhal received the Ph.D. degree in computer science from Ohio State University. He is a Senior Computer Scientist with the Computer Security Division, National Institute of Standards and Technology. His research interests include network security and forensics.

John Yen received the Ph.D. degree in computer science from the University of California at Berkeley in 1986. He is currently a University Professor and the Director of strategic research initiatives with the College of Information Sciences and Technology, Penn State University. His research interests include artificial intelligence, knowledge representation, and machine learning.