Seclius: An Information Flow-based, Consequence-centric Security Metric

Saman A. Zonouz, Robin Berthier, Himanshu Khurana, William H. Sanders, and Tim Yardley

Abstract—It is critical to monitor IT systems that are part of energy delivery systems infrastructure. The problem with intrusion detection systems (IDSes) is that they often produce thousands of alerts daily that must be dealt with by administrators manually. To provide situational awareness, detection systems usually employ (alert, priority) mappings that are either built in the IDS without consideration of the high-level mission objectives of the infrastructure, or manually defined by administrators through a time-consuming task that requires deep system-level expertise. In this paper, we present *Seclius*, an online security evaluation framework that translates low-level IDS alerts into a high-level system security measure and provides a ranking of past malicious events and affected system assets based on how crucial they are for the organization. Seclius significantly reduces human involvement by automatically learning system characteristics, providing a simple formalism that administrators can use to define security requirements. Experiments on a process control network with real vulnerabilities and a multi-step attack show that Seclius can accurately report system security with low performance overhead and support the time-constrained security decision-making process that is necessary for critical infrastructure.

Index Terms—Intrusion Detection Systems, System Security Metric, Information Flow-based Analysis.

1 Introduction

Keeping systems and networks secure is a continual race against attackers. The growing number of security incidents [34] indicates that current approaches to building systems do not sufficiently address the increasing variety and sophistication of threats and block attacks before systems are compromised. Therefore, organizations must resort to trying to detect malicious activity that occurs, so efficient intrusion detection systems (IDSes) [2] are deployed to monitor the systems and identify misbehavior. However, IDSes alone are not sufficient to allow operators to understand the security state of their organization, because monitoring sensors usually report all potentially malicious traffic without regard to the actual network configuration, vulnerabilities, and mission impact. Moreover, given large volumes of network traffic, IDSes with even small error rates can overwhelm operators with false alarms. Even when true intrusions are detected, the actual mission threat is often unclear, and operators are unsure as to what actions they should take. In fact, to respond effectively to system compromises, security administrators need to obtain updated estimate summaries regarding the security status of their mission-critical assets precisely and continuously, based on alerts that occur, in order to prioritize their response and recovery actions. This requirement is even stronger in the context of cyber-physical energy delivery systems infrastructures, in which a cyber-side malicious activity can have catastrophic physical consequences, e.g., the Trans-Siberian pipeline explosion due to a computer trojan horse [30] or the Stuxnet computer worm [8] which was specifically designed to compromise the programmable logic controllers used in nuclear power plants.

The current techniques for the security-state estimation problem generally fall short in two major respects. First, existing solutions rely heavily on human knowledge and involvement [29]. The system administrator must observe the triggered IDS alerts (possibly in a visual manner) and manually evaluate their criticality, which can depend on the alerts' accuracy, the underlying system configuration, and high-level security requirements. As the size of computer networks increases, the manual inspection of alerts usually becomes very tedious, if not impossible, in practice. Requiring extensive human knowledge, the current model-based approaches [21] try to compute security metrics based on a manually designed model and a strong set of assumptions about attackers' behaviors and the vulnerabilities within the system.

Second, previous techniques for IDS alert correlation and system security state estimation usually focus only on the attack paths [13], [46] and subsequent privilege escalations [21], without considering dependencies between system assets. In doing so, they define the security metric of a given system state to be the least required number of vulnerability exploitations (i.e., privilege escalations) needed to get from that state to the goal state in which the attacker gains the privileges necessary to cause his or her final malicious consequence, e.g., a sensitive file modification. We call such metrics *attack-centric* metrics. Therefore, regardless of the transitions, this type of metric is not itself defined for a non-goal state. Equivalently, attack metric definitions are created with the assumption that all attackers will pursue exploitations until they get to the goal state, which is insecure by definition. However, in practice, there are often unsuccessful attacks that cause partial damage to systems, such as a Web server crash as the result of an unsuccessful buffer overflow exploitation. Hence, it is important to consider not only future vulnerability exploitations, but also the damage already caused by the attacker.

To address those different limitations, we introduce an information flow-based system security metric called *Seclius*. Seclius works by evaluating IDS alerts received in real-time to assess how much system and network assets' security has been affected by attackers. This online evaluation is performed using two components: 1) a *dependency graph*, and 2) a *consequence tree*. These two components are designed to identify the context required around each IDS alert to accurately assess the security state of the different assets in the organization.

- *S. Zonouz is with the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL, 33143. E-mail: s.zonouz@miami.edu*
- *R. Berthier, W. Sanders and T. Yardley are with the Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 61801. Emails: {rgb, whs, yardley}@illinois.edu*
- *H. Khurana is with the Integrated Security Technologies division at the Honeywell Automation and Control Systems Lab, Minneapolis, MN 55418. Email: himanshu.khurana@honeywell.com*

Formally, the dependency graph is a Bayesian network automatically learned during a training phase when the system behaves normally. The dependency graph captures the low-level dependencies among all the files and processes used in the organization. The consequence tree is a simple tree structure defined manually by administrators to formally describe at a high level the most critical assets in the organization. When a new IDS alert is received, a belief propagation algorithm, the Monte Carlo Gibbs sampling [4], combines the dependency graph and the consequence tree to calculate in an online manner the probability that the critical assets in the organization have not been affected and are still secure. Consequently, Seclius assesses the overall organizational security using a bottom-up logical propagation of the probabilities that individual assets are or are not compromised.

Seclius minimizes the reliance on human expertise by clearly separating high-level security requirements from the low-level system characteristics of an IT infrastructure. We developed an algorithm and a set of instruments to automatically learn the dependency graph, which represents the system characteristics by capturing information flows between files and processes, within hosts, and across the network. As a result, administrators are not required to define such low-level input, so they can focus entirely on identifying the high-level organizational security requirements, using the consequence tree. These requirements are most often subjective and cannot be automatically discovered. In practice, even in large organizations, the consequence tree contains very few assets, e.g., a Web server and a database, and does not require detailed system-level expertise. In addition, as a *defense-centric* metric, Seclius assesses system security by focusing solely on past consequences; hence, it assumes nothing about system vulnerabilities and attackers' future behaviors, e.g., possible attack paths.

It is worth emphasizing that Seclius does not provide an intrusion detection capability per se; instead, it assesses organizational security based on the set of alerts triggered by underlying IDSes. Therefore, the security measure would not be updated by Seclius if an attack was not detected by an IDS. Furthermore, it is not an intrusion response system and does not automatically respond to attacks, but instead helps administrators or response systems react by providing situational awareness capability.

solely on past consequences, and hence it assumes nothing about system vulnerabilities, and attackers' future behaviors, e.g., possible attack paths.

In summary, the contributions and benefits of this paper are the followings.

- We introduce a defense-centric metric to probabilistically determine if critical assets have been compromised. The metric fills the gap between low-level IDS alerts and the need for comprehensive situational awareness.
- We address the main limitations of existing security metrics by 1) minimizing human involvement and removing assumptions about attackers' behaviors and vulnerabilities, and 2) considering probabilistic dependencies between assets to understand the impact of malicious intrusions.
- We show how Seclius can be used to automatically rank IDS alerts or highlight critical assets that need to be taken care of with high priority.
- We evaluate the performance and accuracy of Seclius in a testbed environment with a prototype framework that we implemented to reproduce a process control environment, and a realistic multi-step attack scenario.
- We compare two existing approaches to generating the system dependency graph, namely instruction-level taint-tracking and the syscall interception, and show that the syscall interception provides a reasonably accurate dependency model while causing low computational overhead on the system.

This paper is organized as follows. We provide an overview of the system in Section 2. We detail the consequence tree and their design process in Section 3, and explain the dependency graph notation in Section 4. The security measure value calculation is described in Section 5. We describe the system's implementation in Section 6 and empirically evaluate it in Section 7. We review the past related work in Section 8 and discuss the Seclius's limitations in Section 9. We finally conclude the paper in Section 10.

2 System Overview

We first describe how attack-centric and defence-centric security metric are distinguished. In past related work on attack-centric security metrics, e.g., attack graph-based techniques, where there is one (or more) goal states, the security measure of a non-goal state in the attack graph is not independently calculated, i.e., it is calculated as a function of "how close" the non-goal state is to the goal state (from an attack-centric viewpoint, how much is left to reach the destination). Therefore, from an attack-centric viewpoint, if the attack stops in a non-goal state, the attacker gains nothing (according to the model); however, from a defense-centric viewpoint, the system may have already got affected through the past exploitations on the attacker's way to get to the current non-goal state. For instance, let us assume that the attacker's end-goal is to read a sensitive file residing in a back-end database server, and on the attacker's way in the current non-goal state, a buffer-overflow vulnerability in a web server system is exploited resulting in the server process crash (unavailable web server). According to the attacker's objective, he has not gained anything yet; however, the defense-centric system security has been affected as a network functionality is lost (web server crash). Consequently, the defense-centric security measure of a non-goal state, independently from the goal state and regardless of whether the attack will succeed, may not be the highest value (1 in our case).

Seclius's high-level goal is to assess the security of each possible system state with minimal human involvement. In particular, we define the security of a system state as a binary vector in which each bit indicates whether a specific malicious event has occurred in the system.

We consider two types of malicious events. First, there are vulnerability exploitations, which are carried out by an attacker to obtain specific privileges and improve his or her control over the system. Therefore, the first set of bits in a state denote the attacker's privileges in that state, e.g., root access on the Web server host. Those bits are used to determine what further malicious damage the attacker can cause in that state. Second, there are attack consequences, which are caused by the attacker after he or she obtains new privileges. Specifically, we defined consequences as the violations of the "CIA" criteria (i.e., Confidentiality, Integrity, and Availability) applied to critical assets in the organization, such as specific files and processes. For example, the integrity of a file F_2, denoted by $I(F_2)$, is compromised if F_2 is either directly or indirectly modified by the attacker.

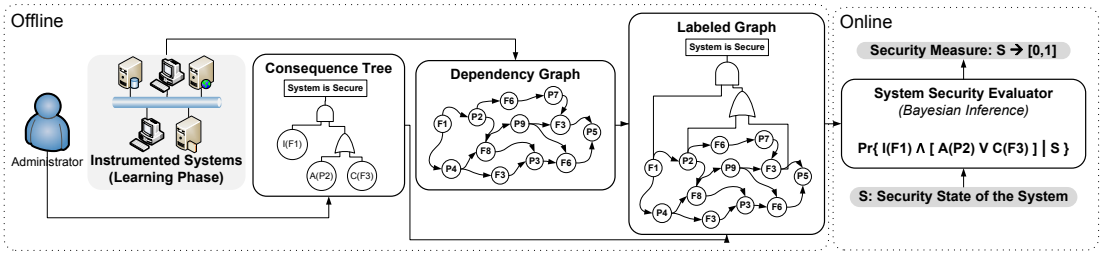


Fig. 1. The Components of the Seclius Framework

According to [37], the security of any given system is characterized by a set of its identifiable attributes, such as security criteria of the critical assets, e.g., integrity of a database file; the notion of a security metric is defined as a quantitative measure of how much of that attribute the system possesses. Our goal is to provide administrators with a framework to compute such measures in an online manner. We believe that there are two major barriers to achieving this goal. First, while the critical assets are system-specific and should be defined by administrators, a framework that requires too much human involvement is prone to errors and has limited usability. As a result, a formalism is needed so that administrators can define assets simply and unambiguously. Second, low-level IDS alerts usually report on local consequences with respect to a specific domain. Consequently, we need a method that provides understanding of what the low-level consequences represent in a larger context and quantifies how many of the security attributes the whole system currently possesses, given the set of triggered alerts.

We developed the Seclius framework (see Figure 1) to address those two challenges. Seclius works based on two inputs. First, a manually defined consequence tree, and an automatically learned dependency graph. The hierarchical and formal structure of the consequence tree enables administrators to define the critical assets easily and unambiguously with respect to the subjective mission of the organization. Second, a dependency graph captures the dependencies between these assets and all the files and processes in the system during a training period. In production mode, Seclius receives low-level alerts from intrusion detection sensors and uses a taint propagation analysis method to evaluate online the probabilities that the security attributes of the critical assets are affected.

Regarding how large a consequence tree can get in a real-world setting, we highlight that the consequence tree needs to include only the main critical assets. For instance, in a banking environment, the credit card database would be labeled as critical. However, the consequence tree does not have to include *indirectly*-critical assets, which are defined as assets that become critical because of their (possible) interaction with other critical assets. For instance, a Web server that interacts with a credit card database would also be partially considered as critical but would not need to be labeled as such. We emphasize that the number of main critical assets is usually very low even for large target infrastructures. All inter-asset dependencies and system-level details are captured by the dependency graph that is generated and analyzed automatically. The small size of the manually-constructed consequence trees and the automated generation of the dependency graphs improve the scalability of Seclius remarkably, as shown in the experiments.

To further clarify the conceptual meaning of the system security measure in Seclius, here we describe in more details what it mathematically represents. The security measure value that Seclius calculates represents the probability that the system is "insecure" according to the consequence tree that is designed by the administrator. For instance, if the consequence tree includes a single node `credit-card-numbers.db` *confidentiality*, the ultimate system security measure value will denote the probability that the confidentiality of the `credit-card-numbers.db` has been compromised at any given state. Consequently the calculated measures range continuously between 0 and 1 inclusively. In our implementations, we used discretization $[0,1] \rightarrow \{$`low`, `medium`, `high` $\}$ to indicate when the system's security is high or when it needs to be taken care of.

To illustrate how Seclius works in practice, consider a scenario in which the IT infrastructure of a power-grid utility is instrumented to enable comprehensive security monitoring of systems and networks. To leverage Seclius on top of the IDSes deployed, administrators would first define critical assets and organizational security requirements through the consequence tree. We emphasize that this task does not require deep-knowledge expertise about the IT infrastructure. In our example, the critical assets would include programmable logic controllers (PLCs), used to control the underlying power system, and the customer database server. In particular, the security requirements would consist of the availability of the PLCs and the integrity and confidentiality of the database files. The second step would be to run a training phase with no ongoing attack in order to collect data on intra- and inter-host dependencies between files and processes. After a few hours, the results of this training phase would be automatically stored in the dependency graph, and the instruments used to track the dependencies would be turned off. The third and final step would be to run Seclius in production mode. Seclius starts processing alerts from IDSes by using the generated dependency graph and probabilistically determines whether the critical assets are compromised. If a vulnerability exploitation in the customer Web server was detected by an IDS, Seclius would update not only the security measure of the corresponding Web server, but also that of the set of systems that depend on the Webserver, e.g., the backend database, according to the learned dependency graph. By observing the real-time IDS alerts and the learned inter-asset dependencies, Seclius can precisely measure 1) the privileges gained by the attacker and which security domains he or she was able to reach, and 2) how the integrity, confidentiality, or availability of the assets has been affected by the exploit directly or indirectly.

In the next section, we further describe the mathematical tools and the formalism used by the various components of Seclius to provide that information.

3 Consequence Tree

We discuss the first manual input required by Seclius, namely the consequence tree (CT). The goal of the CT is to capture critical IT assets and organizational security requirements. The criticality level of individual assets within an organization is indeed an environment-specific issue. In other words, the

criticality levels heavily depend on the organizational missions and/or business-level objectives. For instance, consider a power grid process control environment, whose mission is generation of power and its secure delivery to the customers. In such an environment, provision of high-availability guarantees for a programmable logic controller, which is used to control an important power generator, is often much more critical than for a historian, which is used to store historical sensor measurements for later analyses. Hence, Seclius requires system administrators to manually provide the list of organizational critical assets.

The critical assets could be provided using any function: a simple list (meaning that all items are equally important), a weighted list, or a more complex combination of assets. In this paper, we chose to use a logical tree structure. We believe that it offers a good trade-off between simplicity and expressiveness, and the fact that it can be represented visually makes it a particularly helpful resource for administrators. The formalism of the CT follows the traditional fault tree model [39]; however, unlike fault trees, the leaf nodes of the CTs in Seclius address security requirements (confidentiality, integrity, and availability) of critical assets, rather than dependability criteria.

The CT formalism consists of two major types of logical nodes, namely `AND` and `OR` gates. To design an organizational CT, the administrator starts with the tree's root node, which identifies the main high-level security concern/requirement, e.g., "Organization is not secure." The rest of the tree recursively defines how different combinations of the more concrete and lower-level consequences can lead to the undesired status described by the tree's root node. The recursive decomposition procedure stops once a node explicitly refers to a consequence regarding a security criterion of a system asset, e.g., "availability of the Apache server is compromised." These nodes are in fact the CT's leaf consequence nodes, each of which takes on a binary value indicating whether its corresponding consequence has happened (1) or not (0). Throughout the paper, we use the C, I, and A function notations to refer to the CIA criteria of the assets. For instance, $C(F_2)$ and $I(P_6)$ denote confidentiality of file F_2 and integrity of process P_6, respectively. The leaves' values can be updated by IDSes, e.g., Samhain [43]. The CT is derived as a Boolean expression, and the root node's value is consequently updated to indicate whether the organizational security is still being maintained.

A CT indeed formulates subjective aspects of the system. Its leaf nodes list security criteria of the organization's critical assets. Additionally, the CT implicitly encodes how critical each asset is using the depth of its corresponding node in the tree; that is, the deeper the node is, the less critical the asset is in the fulfillment of the organizational security requirements. Furthermore, the CT formulates redundant assets using `AND` gates. Seclius requires administrators to explicitly mention the redundancies because it is often infeasible to discover redundancies automatically over a short learning phase [5].

Although the CT formulation can be considered as a particular kind of the general attack tree formalism as introduced in [33], its application in Seclius is different from how attack trees have been used typically in the past, as the CTs formulate how past *consequences* contribute to the overall security requirements, whereas attack trees usually address attackers' potential *intents* against the organization. In other words, at each time instant, given the consequences already caused by the attackers in the system, Seclius employs the CT to estimate the current system security, while the system's attack tree is often used to probabilistically estimate how an attacker could or would penetrate the system in the future.

4 Dependency Graph

As mentioned in the previous section, the CT captures only the subjective security requirements, and does not require deep-knowledge expertise about the IT infrastructure, thanks to the dependency graph (DG). The goal of the DG is to free the administrator from providing low-level details about the organization. Those details are automatically captured by Seclius through a learning phase, during which the interactions between files and processes are tracked in order to probabilistically identify direct or indirect dependencies among all the system assets. For instance, in a database server, the administrator only needs to list the sensitive database files, and Seclius later marks the process `mysqld` as critical because it is in charge of reading and modifying the databases. Such a design greatly reduces the resources and time spent by administrators in deploying Seclius.

Each vertex in the DG represents an object, namely a file, a process, or a socket, and the direct dependency between two objects is established by any type of information flow between them. For instance, if data flow from object o_i to o_j, then object o_j becomes dependent on o_i; the dependency is represented by a directed edge in the DG, i.e., $o_i \rightarrow o_j$. To capture that information, Seclius intercepts syscalls and logs them during the learning phase. In particular, we are interested in the syscalls[1] that cause data dependencies among the OS-level objects. A dependency relationship is stored by three elements: a source object, a sink object, and their security contexts. the file system directory tree. When the learning phase is over, syscall logs are automatically parsed and analyzed line by line to generate the DG. Each dependence edge is tagged with a frequency label indicating how many times the corresponding syscalls were called during the execution.

We make use of the Bayesian network formalism to store probabilistic dependencies in the DG; a conditional probability table (CPT) is generated and associated with each vertex. This CPT encodes how the information flows through that vertex from its parents (sources of incoming edges) to its children. For example, if some of the parent vertices of a vertex become tainted directly or indirectly by attacker data, the CPT in the vertex saves the probability that the vertex (specifically, the OS-level object represented by the vertex) also gets tainted.

More specifically, each DG vertex is modeled as a binary random variable (representing a single information flow), equal to either 1 (true) or 0 (false) depending on whether the vertex has been tainted; the CPT in a vertex v stores the probability that the corresponding random variable will take the true value ($v = 1$), given the binary vector of values of the parent vertices $P(v)$. Formally, the probability value is computed using the following equation: $Pr(v|P(v)) = 1 - \prod_{p_v^i \in P(v)} \{1 - \mathbf{1}_{(p_v^i)}.Pr(p_v^i \rightarrow v)\}$ where $\mathbf{1}_{(.)}$ is the indicator function that takes on the value 1 if the condition inside the parentheses holds, and 0 otherwise. $Pr(p_v^i \rightarrow v)$ is the probability that the information will flow from the parent node p_v^i to the vertex v. This probability represents the fraction of times during which information flows from p_v^i to v. It is calculated using the frequency labels on p_v^i's outgoing edges that are captured during the learning phase. In

1. Specifically, we log the following syscalls: `openat` , `dup` , `dup2` , `close` , `write` , `writev` , `read` , `readv` , `ipc` , `clone` , `fork` , `vfork` , `execve` , `open` , `creat` , `mmap` , `mmap2` , `socketcall` , `recv` , `recvfrom` , `recvmsg` , `send` , `sendto` , and `sendmsg` .

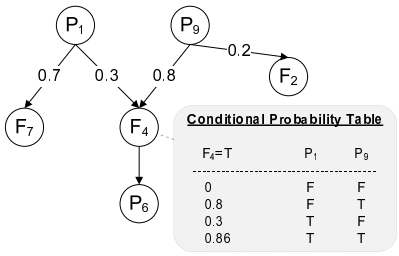


Fig. 2. Conditional Probability Table Construction

summary, the vertex v takes on the value 1 if information flows from any of its parents that have the value 1.

Figure 2 illustrates how a CPT for a single flow (with 1-bit random variables) is produced for a sample vertex, i.e., the file F_4. The probabilities on the edges represent $Pr(. \rightarrow .)$ values. For instance, the process P_1 writes data to the files F_4 and F_7 with probabilities 0.3 and 0.7, respectively. As shown in the figure, the file F_4 cannot become tainted if none of its parents are tainted, i.e., $Pr(F_4|\bar{P}_1,\bar{P}_9) = 0$. If only the process P_1 is tainted, F_4 can become tainted only when the information flows from P_1, i.e., $Pr(F_4|P_1,\bar{P}_9) = 0.3$. If both of the parents are already tainted, then F_4 would get tainted when information flows from either of its parent vertices. In that case, the probability of F_4 being tainted would be the complement probability of the case when information flows from none of its parents. Therefore, $Pr(F_4|P_1,P_9) = 1-(1-0.3)\times(1-0.8) = 0.86$.

Each CT leaf node that represents a CIA criterion of a critical asset is modeled by Seclius as an information flow between the privilege domains controlled by the attacker (according to the current system state) and those that are not yet compromised. Confidentiality of an object is compromised if information flows from the object to any of the compromised domains. Integrity of an object is similarly defined, but the flow is in the reverse direction. Availability is not considered as an information flow by itself; however, an object's unavailability causes a flow originating at the object, because once an object becomes unavailable, it no longer receives or sends out data as it would if it was not compromised. For instance, if a process frequently writes to a file, once the process crashes, the file is not modified by the process, possibly causing inconsistent data integrity; this is modeled as a propagation of tainted data from the process to the file. We consider all the leaf nodes that concern the integrity criterion of critical assets as a single information flow, because they conceptually address the same flow from any of the compromised domains to the assets. However, confidentiality flows cannot be grouped, as they originate individually at separate sources.

If each information flow is represented as a bit, then to completely address n concurrent information flows, we define the random variable in each vertex as an n-bit binary vector in which each bit value indicates whether the vertex is already tainted by the bit's corresponding flow. In other words, to consider all the security criteria mentioned in a CT with n leaves, every vertex represents an n-bit random variable (assuming integrity bits are not grouped), where each bit addresses a single flow, i.e., a leaf node. The CPTs are generated accordingly; a vertex CPT stores the probability of the vertex's value given the value of its parents, each of which, instead of true or false, can take on any n-bit value.

5 System Security Evaluation

Given the DG generated during the learning phase, the operator turns off the syscall interception instruments and puts the system in production mode. The learned DG is then used in an online manner to evaluate the security of any system security state. The goal of this section is to explain how this online evaluation works in detail. We first assume that the IDSes report the exact system state with no uncertainty. We discuss later how Seclius deals with IDS inaccuracies.

At each time instant, to evaluate the security of the system's current state s, DG vertices are first updated according to s, which indicates the attacker's privileges and past consequences (CT's leaf nodes). For each consequence in s, the corresponding flow's origin bit in DG is tainted. For instance, if file F_4 is modified by the attacker (integrity compromise), the corresponding source bit in DG is set to 1 (evidence bit).

The security measure for a given state s is defined to be the probability that the CT's root value is still 0 ($Pr(\texttt{root}_{\texttt{CT}} \mid s)$), which means that the organizational security has not yet been compromised. More specifically, if the CT is considered as a Boolean expression, e.g., $\texttt{CT} = (C(F10) \wedge A(P_6)) \vee I(F_2)$, Seclius calculates the corresponding marginal joint distribution, e.g., $Pr[(C(F10) \cap A(P_6)) \cup I(F_2)]$, conditioned on the current system state (tainted evidence vertices).

Seclius estimates the security of the state s by calling a belief propagation procedure, namely, the Gibbs sampler [4], on the DG to probabilistically estimate how the tainted data (evidence bits) are propagated through the system while it is in state s.

Generally, the Gibbs sampler algorithm [4] is a Monte Carlo simulation technique that generates a sequence of samples from a joint probability distribution of two or more random variables $X_1, X_2, \ldots, X_n$. The purpose of such a sequence in Seclius is to approximate the joint distribution numerically using large number of samples. In particular, to calculate a joint distribution $Pr(X_1, X_2, \ldots, X_n | e_1, \ldots, e_m)$, where e_i represents an evidence, the Gibbs sampler runs a Markov chain on $X = (X_1, X_2, \ldots, X_n)$ by 1) initializing X to one of its possible values $x = (x_1, x_2, \ldots, x_n)$; 2) picking a uniformly random index i ($1 \leq i \leq n$); 3) sampling x_i from $Pr(X_i|x,e)$ (represented by the conditional probability tables in the generated Bayesian network), 4) updating the x vector, and 5) going back to step 2. It has been proved that the stationary distribution of the Markov chain is just the sought-after joint distribution [4]. Thus, drawing samples from the Markov chain at long enough intervals, i.e., allowing enough time for the chain to reach the stationary distribution, gives independent samples from the distribution $P(X_1, \ldots, X_n | e)$.

We make use of the Gibbs sampler algorithm in Seclius for two main reasons. First, the DG model's joint distribution is not explicitly known initially, and second, analytical calculation of it can be tedious, if possible, specially for large DG graphs [4]. The Gibbs sampler uses the DG's CPT to generate a large number of samples from the $Pr[\texttt{CT}|s]$ distribution without directly calculating the density function [4]. Similarly, the security measure is estimated individually for each system state. Therefore, if the attacker modifies any other object and/or gets more privileges, the system would switch to a new state, whose security measure would be separately evaluated.

It is worth emphasizing that Seclius does not use the DG model to estimate how the attacker contacts other objects from a compromised object, such as a tainted process, to exploit a vulnerability and/or escalate his or her privileges. Seclius uses the DG only to estimate how the tainted data would propagate

through other *non-compromised* system assets, which would behave normally as they did during the learning phase. For every asset already compromised, Seclius assumes a pessimistic behavior model, i.e., the asset deterministically contacts all other assets in its privilege domain.

That approach can evaluate the security of each system state. However, the exact current security state of the system is usually not completely observable, due to IDS inaccuracies, i.e., false positive and negative rates. We define the notion of the *information state* of the system, which formally is a probability distribution over all states in the state space of the system $s \in S$. The information state of the system is estimated, based on the IDS alerts and the false positive and negative rates of the IDSes, using the following equation: $Pr(s) = \prod_{i=1}^{|s|-1}(\mathbf{1}_{s_i=1}.[1-\texttt{FP}(s_i)] + \mathbf{1}_{s_i=0}.[1-\texttt{FN}(s_i)])$, where $\mathbf{1}$ is the indicator function, and s_i is the binary state variable in state s. $\texttt{FP}(s_i)$ and $\texttt{FN}(s_i)$ denote the false positive and negative rates, respectively, that depend on the intrusion detection system by which the corresponding alerts are triggered. The false positive and negative rates can be set to be deterministic, i.e., 0, if the detectors are quite accurate. Seclius can also take qualitative values, i.e., $\texttt{FP} : s_i \rightarrow \{\texttt{low}, \texttt{medium}, \texttt{high}\}$, which are later translated into crisp values, i.e., $\{0.25, 0.5, 0.75\}$.

Once the information state of the system has been estimated, Seclius computes the expected security measure of the information state using the following equation: $\sum_{s \in S}(Pr(s).\texttt{Sec}(s))$, where the `Sec` function is the security measure evaluated for the exact state s, as described above.

6 Implementation

Seclius uses syscall interception to capture information flow and learn data dependencies. We implemented it as a loadable kernel module, which currently works with Linux kernels up to 2.6.32-22. However, for recent kernels, before replacing the syscall table entries with our wrapper syscall function pointers, we had to automatically find the address of `sys_call_table`, because its address is no longer exported. Furthermore, we had to make the corresponding memory pages writable as, for security purposes, the syscall table is read-only by default. Once deployed, the module logs called syscalls and information about the calling processes, e.g., PIDs.

Instead of syscall interception, a byte-by-byte information flow tracer, e.g., the TEMU [36] instruction-level taint tracker, could be used. However, TEMU's overhead is usually too high for a production system (see Section 7). However, we used TEMU as a ground truth to evaluate the accuracy of the flow tracer in Seclius. To actually use TEMU in our experiments, we modified it to produce higher-level information (not only instructions) regarding file-system objects, such as files that are being dynamically tainted. Using The Sleuth Kit (TSK) [3], our implementations read the virtual machine's file system and dynamically translate disk-level tainted addresses (generated by TEMU) to file system object names, such as file names and their absolute addresses.

Seclius employs the DlibC++ library [6], which implements the `bayesian_network_gibbs_sampler` object to perform an approximate Monte Carlo inference (using the Gibbs Sampler algorithm) on the DG graph. Briefly, DlibC++ starts by calling the `set_node_value` function and setting the DG node that represent the taint source to true. Then, it implements the Gibbs sampler taint propagation algorithm by randomly sampling possible values from the network, and consequently calculates the desired metric values using the `node_value` function.

7 Evaluations

We now evaluate the accuracy and performance of the various components of Seclius. We designed a set of experiments to empirically answer the following questions: how accurate are the DG-generating instruments, and how much overhead do they generate? How long should the training phase be to build the DG with sufficient coverage? How much time does Seclius require to process alerts and assess the security metric value of the different assets? And finally, how does an inaccurate CT affect the results? We start in this section by describing the experimentation setup, and then proceed to examine the first five questions related to accuracy and performance.

Experimentation Setup. The experiments in this section were conducted on a virtual machine so that we could easily reset the testing environment to a clean state. The host computer had a 2.20 GHz AMD Athlon™64 3700+ CPU, 1 MB of cache, and 2.0 GB of RAM. The guest OS, using VirtualBox, was running Ubuntu 9.04 with a Linux 2.6.22 kernel. We installed several applications, including OpenSSH server, Apache2, MySQL, the eVision content management system, and the RoomPHPlanning Web scheduling application. We additionally changed the `php.ini` file to make the system vulnerable to PHP code injection attacks.

To feed intrusion detection alerts to Seclius, we installed the following sensors: Snort [31]: a lightweight network-based IDS that uses a database of known attack signatures. In the experiments, we ran Snort in fast-alerting detection mode with its standard 74 rule-sets; DazukoFS [23]: a cross-platform device driver that allows applications to control file accesses on the system; ClamAV [16]: an open-source antivirus toolkit. In particular, we used Sigtool in ClamAV to update the virus signatures and Clamscan for scanning downloaded suspicious files. Samhain [43]: a host-based IDS that periodically checks file integrity; Zabbix [49]: a statistical anomaly-based IDS to detect process or network failures and resource-intensive activities; and PHPIDS [27]: a monitoring tool that PHP applications can use to detect cross-site-scripting and remote code execution attacks.

Accuracy and Overhead of the DG Instruments. As mentioned in the previous section, we developed a syscall interception kernel module to automatically generate the DG. Seclius also captures detailed security contexts through SE-Linux, which we enabled and configured with the default Ubuntu policy `policy.24`. The experiments were done on the Apache Web server process while it was benchmarked from a remote machine, using the command `ab -n 50000`, which subsequently generate repeated HTTP requests. Figure 3 shows a sample DG for the experimental Web server; rectangles in the figure represent different SE-Linux privilege domains within the system, and the vertices are OS-level objects. For clarity, only the process nodes are shown. The original graph, including all FIFOs and other OS-level objects, consisted of 8,396 nodes[2]. According to the syslog logs, the process had periodically been reading from the network sockets `SOCK_AF_INET`, and then accessed some FIFO pipes, the `/var/www/index.html` and `/var/log/apache2/access.log` regular files. Figure 4(a) shows the data-flow histogram of the DG. More specifically, the vertical axis depicts how many vertex pairs in the DG experienced a specific data-flow frequency given by the horizontal axis in the figure.

2. As discussed earlier in the paper, automated solutions are used in Seclius to analyze the dependency graphs and the graphs with real-world sizes can be processed efficiently as shown by the experiments.

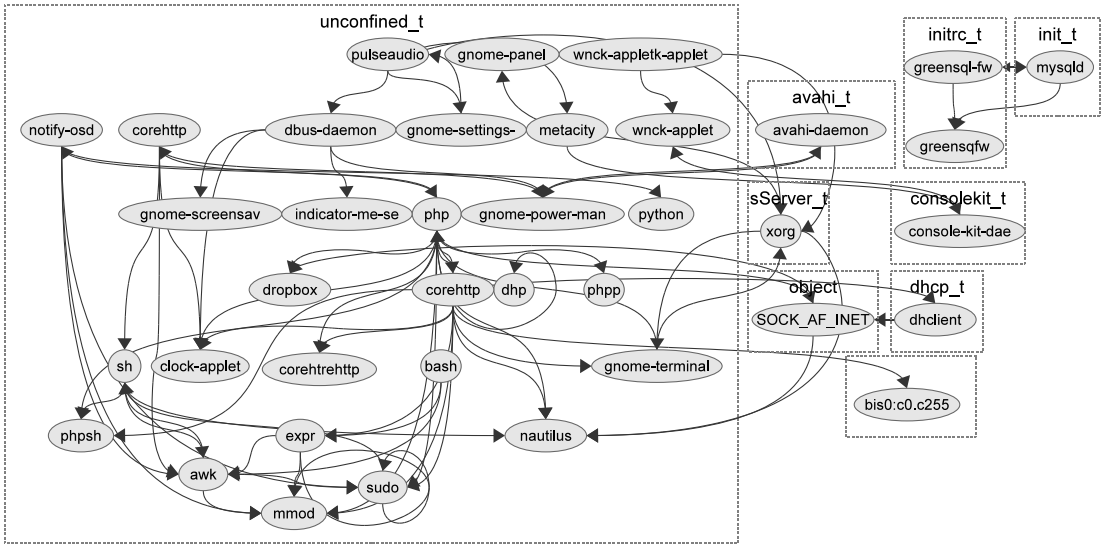
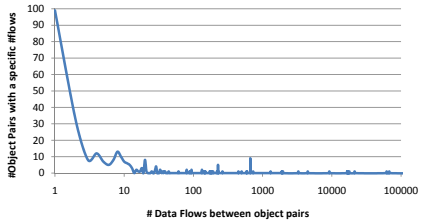
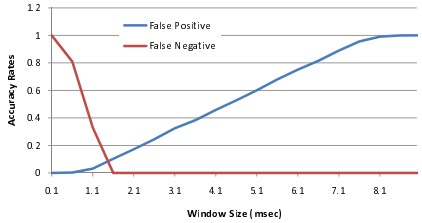


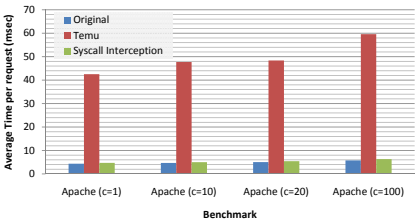
Fig. 3. Dependency Graph Including Processes Only



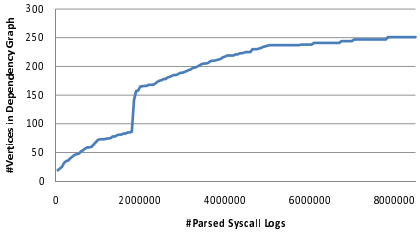
(a) The DG's Behavior Histogram



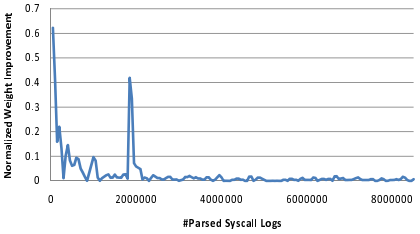
(b) Accuracy of Syscall Taint Tracking



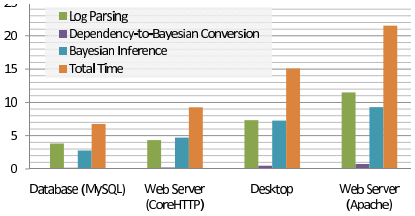
(c) Overhead Syscall vs. TEMU



(d) Dependency Graph Convergence



(e) DG's Edge Label Convergence



(f) Time Requirements

Fig. 4. Convergence and Accuracy of Dependency Graph Generation and Analysis

As mentioned in the previous section, we used the byte-by-byte taint tracking engine TEMU as ground truth to evaluate the DG's accuracy. We define the DG's accuracy as the portion of dependencies correctly identified among system assets after comparison with TEMU's results. Inaccuracies are due to the sliding window of size t used by our approach to associate a DG vertex's outgoing data flow with each incoming input. As a result, the numbers of false positives and false negatives depend directly on the size of t. We varied t when tracking the Apache process and compared the results to TEMU's. The false positive rate was calculated as $\#FalsePositives/(\#FalsePositives+\#TrueNegatives)$. Similarly, the false negative rate was estimated using $\#FalseNegatives/(\#FalseNegatives+\#TruePositives)$. Figure 4(b) shows the results for the various window sizes employed. The false positive rate grows linearly up to 1.0 as the window size exceeds 8.1 milliseconds. The false negatives, on the other hand, quickly drop to zero before the window size gets to 2 milliseconds. Based on these results, 1.5 milliseconds seems to be a reasonable window size for our environment, as both the false positive and negative rates remain fairly low.

We then proceed to evaluate the overhead of the syscall tracking module, still using TEMU as a benchmark. Figure 4(c) shows how much overhead each technique caused within the system in terms of end-point response time for the Apache Web server. The server was benchmarked against different numbers of concurrent clients. As shown in the figure, the syscall interception caused less than 10% overhead on average compared to the original execution, which was not instrumented, whereas the TEMU engine caused an approximately 850% slowdown in the Web server response time, which is clearly not acceptable for practical uses.

In conclusion, our choice to use syscall tracking to generate the DG offers a good tradeoff between accuracy and overhead. Once properly configured, it has less than 0.1% false positives and no false negative while providing under 10% overhead. We note that this overhead occurs only during the training phase and disappear when the system runs online in production.

Required Training Time for Sufficient Dependency Coverage. To configure Seclius, the normal system behavior (i.e., the DG) should be captured and learned. The training duration should be long enough to create a model representing the actual system. On the other hand, to accelerate the setup phase of Seclius and avoid large syscall logs, the training mode should be reasonably short. In this section, we evaluate how long it takes for the DG parameters to converge in the context of the Apache Web server running a PHP application, namely the eVision content management system.

First, we focused on how the DG cardinality, i.e., number of vertices, gets updated and finally converges. We initially set the system up for the training without any incoming Web client requests. Figure 4(d) shows the DG cardinality during the experiment vs. the number of syslogs, i.e., 1.8E6 syscalls per minute on average. The DG cardinality approached 87 within 67 seconds. Then, we started the `ab` application on a remote machine to send subsequent requests to the server; the request flow caused a 64-vertex jump in the DG cardinality. The reason for the jump was that Apache started accessing other OS-level objects, such as `/var/www/eVision/index.php` . It took approximately 14 more minutes for the DG cardinality again to converge completely to its stable value. The whole learning phase took about 19 minutes, and approximately 2.3 GB of syscalls were logged.

Second, we evaluated how the DG edge labels, i.e., probability values, converged during the training mode. Figure 4(e) shows the normalized label updates, i.e., the difference between the last and the updated values, while syslogs were being parsed. As shown in the figure, it took about 3 minutes on average for the edge labels to converge. The launched request flow caused a spike in the graph, but the normalized label updates quickly converged back to zero again.

In summary, the time required to generate the DG is heavily dependent on the applications analyzed and their usage but our experiment in the context of a typical Web server shows that 20 minutes is sufficient.

Online Processing Time to Generate Security Metric Values. Once the training mode is complete, Seclius translates the DG to a Bayesian network, and then uses it to evaluate the system security in an online manner. In this section, we evaluate how much each step of the algorithm contributes to the overall processing time.

The four major automated steps conducted by Seclius are 1) syscall interception (the learning phase), 2) parsing of the syslogs and the DG creation, 3) conversion of the DG to the Bayesian network (CPT generation), and 4) online security evaluation via implementation of the Gibbs sampler. The learning phase was discussed in detail in the previous two experiments. Figure 4(f) shows the time required for the steps after the system model has been learned on four machines. As demonstrated, parsing of syscall logs (43% on average) and the online security evaluation (53% on average) constitute two major portions of the total time, while the translation to the Bayesian network takes less than 4% of the total time.

Like any other Monte Carlo-based sampling technique, the Gibbs sampler algorithm terminates once the iterative conditional probability estimation improvement has satisfied a convergence threshold. Figure 5(a) shows how the delta value, i.e., the absolute value of the improvement between two subsequent sampling iterations, approaches zero (the Y axis is shown in the logarithmic scale base 10) with respect to the finished iterations, i.e., approximately 1,200 iterations per second.

In conclusion, the online processing time of Seclius to process alerts and assess the security metric value of the different assets is measured in seconds, ranging from 7 seconds for a database up to 21 seconds for a complex Web server.

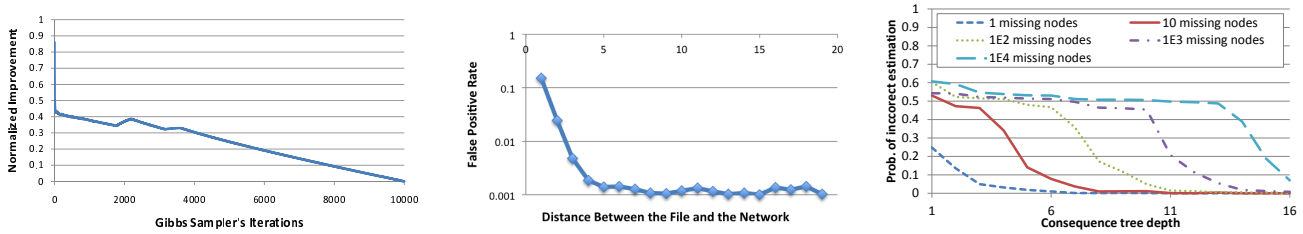
False positive rate analysis. To evaluate the accuracy of the security metric calculation module in Seclius, we measured the false positive rate of the results in various situations. We used the TEMU byte-by-byte information flow tracking solution results as the ground truth and compared them to the results by Seclius. In particular, we ran several scenarios where the sensitive file had not been actually modified by the data from the attacker site (privilege domain). During the experiments, we counted the number of times where Seclius by mistake marked the sensitive file as modified maliciously. As discussed earlier, Seclius may make such a mistake as it tracks information via a time-sensitive system call sequence analysis rather than the heavy-weight byte-by-byte analysis approach like in TEMU. Such a system call sequence-based information flow analysis may identify some data flows from the attacker domain to the domain where the sensitive file resides by mistake while TEMU does not report any such file integrity violation. In our experiments, we fixed the attacker privilege domain, i.e., the outside network by default, and we put the sensitive file in a different non-compromised privilege domain during each experimental scenario. Due to the nature of Seclius and how the security measure values are calculated, when there is no on-going attack within the system, i.e., there is only a single compromised privilege domain in the outside network, the system security measure is evaluated very close to absolute but not 100%. Figure 5(b) shows the corresponding results. The vertical axis shows the false positive rate, and the horizontal axis denotes the shortest distance between the dependency graph node corresponding to the only compromised domain and the node corresponding to the sensitive file privilege domain. As shown, the false positive rate diminishes very quickly when the file's node in the dependency graph is not very close to the nodes that represent sockets for external network connections.

Impact of Inaccurate Security Requirements. The main purpose of this experiment was to evaluate the sensitivity of the Seclius to human errors. In particular, all of the steps, except the consequence tree design, within the Seclius framework are accomplished automatically and the models are extracted from the system. Therefore, the main point in which a mistake may be made is when the system security administrators design the consequence trees manually (even though the design does not require deep system-level expertise). We were interested in evaluating how much the accuracy of the CT impact estimation of the security state. For a given CT and for various leaf node values, we counted how many times the CT's root value was updated. In particular, we compared situations in which the full tree was used to get the root node value to situations in which the tree was partially missing randomly chosen nodes. We ran this comparison for a large number (1,000) of randomly generated full binary CTs in which nodes between the leaves and the root were chosen to be either `AND` or `OR` gates with a probability of 0.5. We did the experiment for full trees of a depth between 1 and 16 (Note that to cover a wide spectrum of consequence trees, depth of some of the randomly-generated trees are artificially large here.) In each case, we compared the results when various numbers of nodes were missing.

Figure 5(c) shows the results for different scenarios. When the depth of the generated tree approaches 16, i.e., 2^{16} leaf nodes, the probability of getting an incorrect security estimation goes to zero for a fixed number of missing nodes. Note that when the number of missing nodes, shown on the graph, is larger than the tree order, all nodes are in fact missing; in other words, security is estimated without use of the CT, and hence, the estimated security is correct with an approximately 0.5 chance.

8 Related Work

Extensive research has been conducted over the past decade on the generic topics of system situational awareness [38], [44],



(a) Convergence of the Gibbs Sampler Algorithm (b) False Positive Rate Analysis (c) Incorrect Estimation Due to Incorrect CT

Fig. 5. Convergence and Uncertaintly Impact Analysis

[47] and security metrics [7], [19], [45]. In this section, we survey the literature most related to our approach and discuss how Seclius contributes to advance the state-of-the-art.

Second, there are dynamic methods, most of which are based on attack graph analysis [13], [21]. The main idea is to capture potential system vulnerabilities, and then extract all possible attack paths. The generated graph can be used to compute security metrics [17], [22], [40], [42], assess the security strength of a network [26], [41], to identify the most critical assets in the organization [32], or for security visualization [20]. They can also be used predictively to rank IDS alerts. In particular, [22] uses an approach called Topological Vulnerability Analysis (TVA) [11], [12] to match network configuration with attack simulation in order to optimize IDS sensor placement and to prioritize IDS alerts. The main issue with attack-graph-based techniques is that they require important assumptions about attacker capabilities and vulnerabilities [9], [35]. There have been several efforts to take into account unknown vulnerabilities during the system security analysis. Zhang et al. [50] evaluate the idea of predicting unknown vulnerabilities in well-known software applications through analysis of the past available historical data set about their know vulnerabilities and the corresponding statistics. The authors conclude that the vulnerabilities do not follow a particular pattern and hence are not predictable accurately. Ou [25] introduces a logic-based hypothetical analysis of what system components would be affected if a particular currently-unknown vulnerability existed in the system. Moreover, the rationale behind NetSPA by Ingols et al. [10] is that although it is impossible to predict the existence of any specific zero-day vulnerability, it is possible to hypothesize a zero-day vulnerability in specific software applications to ensure that the impact of an eventual zero-day can be understood and minimized. The hypothetical analysis of unknown vulnerabilities presents a great solution to investigate the impact of a vulnerability in any specific point in the system; however, a complete system security analysis would require hypothesizing vulnerabilities in every possible system point that is not an optimally scalable solution specially in large-scale infrastructures. While these approaches are important in planning for future attack scenarios, we take a different perspective by relying on past consequences, actual security requirements, and low-level system characteristics, such as file and process dependencies, instead of hypothetical attack paths. As a result, our method is defense-centric rather than attack-centric and does not suffer from the issues of unknown vulnerabilities and incomplete attack coverage.

Other important defense-centric approaches include M-Correlator [28] and FuzMet [1]. These techniques use manually filled knowledge bases of alert applicability, system configuration, or target importance to associate a context with each alert and to provide situational awareness accordingly. Seclius offers a more practical solution by minimizing and simplifying the required manual inputs. It does so by learning low-level system characteristics automatically in order to evaluate precisely the extent to which alerts affect the critical components of the organization.

Such a damage assessment feature has previously been explored via file-tainting analysis for malware detection [48], for offline forensic analysis using backtracking [14], [15] or forward-tracking techniques [51], and for online damage situational awareness [18]. At the network level, [24] introduced a vulnerability definition language (OVAL) and generate a network dependency graph to measure the potential impact of vulnerabilities. In [18], information flow is tracked across multiple layers, namely at the instruction level and at the OS process level. Compared to our approach, this cross-layer technique is more precise, but it requires implementation in a virtual environment and can cause major performance degradation.

9 Discussion

We discuss current limitations of Seclius and potential solutions to address each of them.

First, as in any learning algorithm, it is not guaranteed that the learned DG actually captures every single dependency. One trivial solution would be to make sure that the learning phase is long enough to capture all the dependencies. Alternatively, an active learning algorithm could be used. For instance, the configuration files could be parsed to extract potential dependencies, or a mechanism to make sure all the program paths are traversed. Replacing passive learning with an active algorithm would require application-specific knowledge; however, it would help to accelerate the learning phase.

Second, the evaluated security value will be affected by the accuracy of the underlying intrusion detection solutions, i.e., if some malicious events are missed by the intrusion detectors. Our main contribution in this paper is in showing how to make use of the system dependency graph and the security requirements to evaluate the security of any *given* state; in other words, we do not claim to have come up with a new intrusion detection technique. However, our tool, which makes use of Seclius to evaluate system security, takes under consideration the intrusion detection inaccuracies, i.e., false positive and negative rates, if provided. Additionally, security evaluation by Seclius is done based on the past consequences, which are easier to detect than exploitations. As a case in point, detecting that the Web server is unavailable is usually simpler than determining the exploitation that caused the server crash.

Additionally, as Seclius is an information flow-based metric, when the system has not yet been attacked, Seclius usually evaluates the system security to be close to absolute, but not 100% secure. This is because even during the system's normal operational mode, information is often flowing from external end points, where attackers potentially reside, to critical assets. A possible solution to this problem would be to normalize the evaluated security measure based on the measure of the non-compromised system.

10 Conclusion

We proposed *Seclius*[3], an online security evaluation framework that leverages dependencies between OS-level objects to measure the probability that critical assets have been directly or indirectly compromised. The different components of our framework address three important limitations faced by traditional security evaluation techniques. First, a consequence tree captures the subjective security requirements and minimizes administrator input. Second, Seclius processes IDS alerts online to measure actual attack consequences and does not rely on assumptions about attacker behaviors or system vulnerabilities. Third, a dependency graph is combined with a taint-tracking method to probabilistically evaluate the system-wide impact of locally detected intrusions as well as attacker privileges and security domains, without making assumption about attack paths. Experiments showed that Seclius efficiently learns the system's normal behavior with only 6% overhead, and evaluates the system security within 10 seconds. Finally, we demonstrated how Seclius can be applied in a control system environment to provide security administrators with a comprehensive situational awareness solution.

References

[1] K. Alsubhi, E. Al-Shaer, and R. Boutaba. Alert prioritization in intrusion detection systems. In *IEEE Network Operations and Management Symposium (NOMS)*, pages 33–40, 2008.

[2] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, U.S Department of Energy, 2000.

[3] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[4] George Casella and Edward I. George. Explaining the gibbs sampler. *The American Statistician*, 46(3):pp. 167–174, 1992.

[5] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 117–130. USENIX Association, 2008.

[6] Dlib C++, available at: http://dlib.net, 2010.

[7] R. English and R. Poet. Towards a metric for recognition-based graphical password security. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 239 –243, sept. 2011.

[8] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, October 2010.

[9] N. Idika and B. Bhargava. Extending attack graph-based security metrics and aggregating their application. *Dependable and Secure Computing, IEEE Transactions on*, 9(1):75 –85, jan.-feb. 2012.

[10] Kyle Ingols, Matthew Chu, Richard Lippmann, Seth Webster, and Stephen Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 117–126. IEEE, 2009.

[11] S. Jajodia and S. Noel. Topological vulnerability analysis: a powerful new approach for network attack prevention, detection, and response. *Indian Statistical Institute Monograph Series*, 2008.

[12] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. *Managing Cyber Threats*, pages 247–266, 2005.

[13] S Jha, O Sheyner, and J Wing. Two formal analyses of attack graphs. *Ruan Jian Xue BaoJournal of*, 21(4):49–63, 2002.

[14] Xuxian Jiang, Florian P. Buchholz, Aaron Walters, Dongyan Xu, Yi-Min Wang, and Eugene H. Spafford. Tracing worm break-in and contaminations via process coloring: A provenance-preserving approach. *IEEE Trans. Parallel Distrib. Syst.*, 19(7):890–902, 2008.

[15] S.T. King and P.M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems (TOCS)*, 23(1):51–76, 2005.

[16] Tomasz Kojm. Clamav, available at http://www.clamav.net/, 2009.

[17] I. Kotenko and M. Stepashkin. Attack graph based evaluation of network security. In *Communications and Multimedia Security*, pages 216–227. Springer, 2006.

[18] P. Liu, X. Jia, S. Zhang, X. Xiong, Y.C. Jhi, K. Bai, and J. Li. Cross-layer damage assessment for cyber situational awareness. In *Cyber Situational Awareness*, volume 46, pages 155–176. Springer US, 2010.

[19] A.A. Neto and M. Vieira. Untrustworthiness: A trust-based security metric. In *Risks and Security of Internet and Systems (CRiSIS), 2009 Fourth International Conference on*, pages 123 –126, oct. 2009.

[20] S. Noel, M. Jacobs, P. Kalapa, and S. Jajodia. Multiple coordinated views for network attack graphs. In *IEEE Workshop on Visualization for Computer Security (VizSEC)*, pages 99–106, 2005.

[21] Steven Noel and Sushil Jajodia. Optimal ids sensor placement and alert prioritization using attack graphs. *Journal of Network and Systems Management*, 16(3):259–275, 2008.

3. This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097.

[22] Steven Noel and Sushil Jajodia. Optimal ids sensor placement and alert prioritization using attack graphs. *J. Netw. Syst. Manage.*, 16:259–275, September 2008.

[23] John Ogness. Dazuko: An open solution to facilitate on-access scanning. In *Proceedings of the 13th Virus Bulletin International Conference*, pages 1–5, 2003.

[24] X. Ou, S. Govindavajhala, and A.W. Appel. MulVAL: A logic-based network security analyzer. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*, page 8. USENIX Association, 2005.

[25] Xinming Ou and Andrew W Appel. *A logic-programming approach to network security analysis*. Princeton University, 2005.

[26] J. Pamula, S. Jajodia, P. Ammann, and V. Swarup. A weakest-adversary security metric for network configuration security analysis. In *Proceedings of the 2nd ACM workshop on Quality of protection*, page 38. ACM, 2006.

[27] Phpids available at http://php-ids.org/, 2010.

[28] P.A. Porras, M.W. Fong, and A. Valdes. A mission-impact-based approach to INFOSEC alarm correlation. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 95–114. Springer-Verlag, 2002.

[29] Marcus J. Ranum, Kent Landfield, Michael T. Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the 11th Conference on Systems Administration (LISA '97)*, pages 1–8, Berkeley, CA, USA, 1997. USENIX Association.

[30] T. Reed and G. Bush. *At the Abyss: An Insider's History of the Cold War*. Random House Publishing Group, 2005.

[31] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, pages 229–238, 1999.

[32] R. Sawilla and X. Ou. Identifying critical attack assets in dependency attack graphs. *Computer Security-ESORICS 2008*, pages 18–34, 2008.

[33] Bruce Schneier. Attack trees. *Dr. Dobb?s journal*, 24(12):21–29, 1999.

[34] A. Sharma, Z. Kalbarczyk, R. Iyer, and J. Barlow. Analysis of credential stealing attacks in an open networked environment. In *Proceedings of the 4th International Conference on Network and System Security (NSS)*, pages 144–151, 2010.

[35] Anoop Singhal and Xinming Ou. Techniques for enterprise network security metrics. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 25:1–25:4, New York, NY, USA, 2009. ACM.

[36] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information System Security (ICISS)*, pages 1–25, 2008.

[37] SSE-CMM: Systems Security Engineering Capability Maturity Model, Security Metrics, International Systems Security Engineering Association (ISSEA). http://www.sse-cmm.org/metric/metric.asp.

[38] W.W. Streilein, J. Truelove, C.R. Meiners, and G. Eakman. Cyber situational awareness through operational streaming analysis. In *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pages 1152 –1157, nov. 2011.

[39] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Washington, DC, 1981.

[40] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia. An attack graph-based probabilistic security metric. *Data and Applications Security XXII*, pages 283–296, 2008.

[41] L. Wang, S. Noel, and S. Jajodia. Minimum-cost network hardening using attack graphs. *Computer Communications*, 29(18):3812–3824, 2006.

[42] L. Wang, A. Singhal, and S. Jajodia. Measuring the overall security of network configurations using attack graphs. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 98–112. Springer-Verlag, 2007.

[43] Brian Wotring, Bruce Potter, Marcus Ranum, and Rainer Wichmann. *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.

[44] Haidong Xiao and Ning Chen. Analysis of cyberspace security situational awareness based on fuzz reason. In *Intelligence and Security Informatics (ISI), 2011 IEEE International Conference on*, pages 316 –319, july 2011.

[45] Anming Xie, Weiping Wen, Li Zhang, Jianbin Hu, and Zhong Chen. Applying attack graphs to network security metric. In *Multimedia Information Networking and Security, 2009. MINES '09. International Conference on*, volume 1, pages 427 –431, nov. 2009.

[46] Peng Xie, J.H. Li, Xinming Ou, Peng Liu, and R. Levy. Using Bayesian networks for cyber security analysis. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 211–220, 2010.

[47] Vinod Yegneswaran, Paul Barford, and Vern Paxon. Using Honeynets for Internet Situational Awareness. 2005.

[48] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 116–127. ACM, 2007.

[49] Zabbix available at http://www.zabbix.org, 2010.

[50] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

[51] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. pages 217–226. IEEE Computer Society, 2003.

Saman Zonouz is an Assistant Professor in the Electrical and Computer Engineering Department at the University of Miami. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2011. He has worked on intrusion response and recovery, information flow-based security metrics for power-grid critical infrastructures, online digital forensics analysis and monitorless recoverable applications. His research interests include: computer security and survivable systems, control/game theory, intrusion response and recovery systems, automated intrusion forensics analysis, and information flow analysis-based security metrics.

Robin Berthier is a research scientist at the University of Illinois at Urbana-Champaign, working with Prof. William H. Sanders. Robin graduated from the Reliability Engineering Department at the University of Maryland in 2009. His doctoral dissertation with Prof. Michel Cukier, and focused on the issue of honeypot sensors deployed on large networks. He introduced a new architecture to increase the scalability of high-interaction honeypots, and combined network datasets of different granularities to offer unique attack forensics capabilities. His current research interests include advanced intrusion detection systems and the security of critical infrastructures.

Himanshu Khurana is the senior technical manager for the Integrated Security Technologies section at Honeywell Automation and Control Systems Research Labs, where he is currently working on developing security solutions for messaging systems and the electric grid infrastructure, among other things. Before joining Honeywell, Khurana was a principal research scientist at the Information Trust Institute at the University of Illinois, Urbana-Champaign, and served as the co-principal investigator and principal scientist for the Trustworthy Cyber Infrastructure for Power (TCIPG) center. He obtained his MS and PhD from the University of Maryland, College Park.

William Sanders is a Donald Biggar Willett Professor of Engineering and the Director of the Coordinated Science Laboratory (www.csl.illinois.edu) at the University of Illinois at Urbana-Champaign. He is a professor in the Department of Electrical and Computer Engineering and Affiliate Professor in the Department of Computer Science. He is a Fellow of the IEEE and the ACM, a past Chair of the IEEE Technical Committee on Fault-Tolerant Computing, and past Vice-Chair of the IFIP Working Group 10.4 on Dependable Computing. He was the founding Director of the Information Trust Institute (www.iti.illinois.edu) at Illinois. Dr. Sanders's research interests include secure and dependable computing and security and dependability metrics and evaluation, with a focus on critical infrastructures.

Tim Yardley is the Assistant Director for Testbed Services and a senior researcher and lead for many projects in ITI. He works to define the vision and direction for testbed initiatives under ITI and engages in research to address ITI's mission. His work addresses trustworthiness and resiliency in critical infrastructure, with a particular focus on cyber security, and includes analysis and development of techniques for securing components, systems, and networks. Yardley's research in those areas includes work on control systems, telecommunications systems, critical incident response, and simulations of real-world systems.