Scalable Analysis of Attack Scenarios^{*}

Massimiliano Albanese¹, Sushil Jajodia², Andrea Pugliese³,
and V.S. Subrahmanian¹

¹ University of Maryland
{albanese,vs}@umiacs.umd.edu

² George Mason University
jajodia@gmu.edu

³ University of Calabria
apugliese@deis.unical.it

Abstract. Attack graphs have been widely used for attack modeling, alert correlation, and prediction. In order to address the limitations of current approaches – scalability and impact analysis – we propose a novel framework to analyze massive amounts of alerts in real time, and measure the impact of current and future attacks. Our contribution is threefold. First, we introduce the notion of generalized dependency graph, which captures how network components depend on each other, and how the services offered by an enterprise depend on the underlying infrastructure. Second, we extend the classical definition of attack graph with the notion of timespan distribution, which encodes probabilistic knowledge of the attacker’s behavior. Finally, we introduce attack scenario graphs, which combine dependency and attack graphs, bridging the gap between known vulnerabilities and the services that could be ultimately affected by the corresponding exploits. We propose efficient algorithms for both detection and prediction, and show that they scale well for large graphs and large volumes of alerts. We show that, in practice, our approach can provide security analysts with actionable intelligence about the current cyber situation, enabling them to make more informed decisions.

Keywords: Attack graphs, dependency graphs, vulnerability analysis, cyber situation awareness, scalability.

1 Introduction

Attack graphs have been widely used to model the possible ways an attacker can exploit network vulnerabilities, and to correlate alerts. There has been extensive work on automatically deriving attack graphs from network scans [10,11]. However, existing approaches to alert correlation typically have two major limitations. First, attack graphs do not provide a mechanism to evaluate the likelihood of each attack pattern or its impact on the enterprise. Second, although scalable

^{*} The work presented in this paper is supported in part by the Army Research Office MURI award number W911NF-09-1-0525.

generation of attack graphs has been studied [10], scalability issues with respect to the alert correlation process have not been fully addressed yet.

In order to address the above limitations, we propose a novel framework to analyze massive amounts of raw security data in real time, and measure the impact of current and future attacks. First, we introduce the notion of *generalized dependency graph*, which captures how network components depend on each other, and how the services offered by an enterprise depend on the underlying infrastructure. Second, we extend the classical definition of attack graph with the notion of *timespan distribution*, which encodes probabilistic knowledge of the attacker's behavior. Third, we introduce the notion of *attack scenario graph*, which combines dependency and attack graphs, bridging the gap between known vulnerabilities and the missions or services that could be affected by the corresponding exploits. In the last several years, there has been significant work in dependency modeling [13] and automatic discovery of dependencies [3]. However, to the best of our knowledge, we are the first to combine attack and dependency graphs to enhance situation awareness. In practice, the proposed framework provides security analysts with a more complete picture of the cyber situation. In fact, combining attack and dependency graphs not only provides analysts with a set of possible future scenarios, but also estimates their probability and potential impact. Finally, in order to guarantee scalability, we propose efficient algorithms to track and index ongoing attacks and analyze future scenarios, and show that they scale well for large graphs and large volumes of incoming alerts.

Throughout this paper, we will use the network of Figure 1(a) as a running example. This network offers two services, *Online Shopping* and *Mobile Order Tracking*, and consists of three subnetworks delimited by firewalls. Two subnetworks implement the two services, and each of them includes a host accessible from the Internet. The third subnetwork implements the business logic, and includes a central database server. An attacker who wants to steal sensitive data from the main database server will need to breach the firewalls and gain privileges on several hosts before reaching the target. The attack graph of Figure 1(b) shows that, once a vulnerability V_C on the Mobile Application Server has been exploited, we can expect the attacker to exploit either V_D or V_F . However, the attack graph alone does not answer the following important questions: Which vulnerability has the highest probability of being exploited? Which attack pattern will have the largest impact? How can we mitigate the risk? Our framework is designed to answer these questions efficiently.

The architecture of the proposed framework is sketched in Figure 1(c). We assume that attack and dependency graphs are given, and combine them into an attack scenario graph (Section 5). Incoming alerts are matched against such data structure and indexed in real-time.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the notion of *generalized dependency graph*, whereas Section 4 defines *probabilistic temporal attack graphs*. Section 5 introduces the notion of *attack scenario graph*, and Section 6 presents the proposed index data structure, along with an algorithm to update the index in real-time. Section 7 presents an

Fig. 1. Running example and system architecture

algorithm to analyze and rank future possible scenarios. Finally, Section 8 reports experimental results, and Section 9 presents conclusions.

2 Related Work

To reconstruct attack scenarios from isolated alerts, some correlation techniques employ prior knowledge about attack strategies [5] or alert dependencies [17]. Other techniques aggregate alerts with similar attributes [21] or statistical patterns [20]. To the best of our knowledge, the limitation of the nested-loop approach [22], especially for correlating intensive alerts in high-speed networks, has not been addressed. In [19], Noel *et al.* adopt a vulnerability-centric approach to alert correlation, because it can effectively filter out bogus alerts. However, the nested loop procedure is still used in [19]. Attack scenarios broken by missed alerts are reassembled by clustering alerts with similar attributes [18]. Designed for a different purpose, the RUSSEL language is similar to our approach in that the analysis only requires one-pass of processing [8].

With respect to the general problem of modeling activities, Hidden Markov Models and their variants have been used extensively. For instance, Duong et al. [6] introduce the Switching Hidden Semi-Markov Model, a two-layered extension of the Hidden Semi-Markov Model. Dynamic Bayesian Networks [9] and

probabilistic extensions of Petri Nets [1] have also been used for tracking and recognizing multi-agent activities. A survey of temporal concepts and data models used in unsupervised pattern mining from symbolic temporal data is presented in [16]. In recent years, there has been extensive research in Data Stream Management Systems [7]. However, the scope of our work is drastically different. In fact, we are not interested in retrieving sets of data items satisfying certain conditions and updating such sets as new data is received. Instead, we are interested in identifying sets of alerts that, with a probability above a given threshold, constitute the “evidence” that a cyber attack occurred.

In large enterprise networks, the performance of an application may depend on many hosts and network components. Recently, automated discovery of dependencies from network traffic [2,12] has been proposed. Chen et al. [4] present a comprehensive study of the performance and limitations of this class of dependency discovery techniques, and introduce a new system, Orion, that discovers dependencies using packet headers and timing information in network traffic.

To the best of our knowledge, there has been virtually no work on efficient indexing to support scalable and real-time attack detection. Similarly, there has been no work on integration of attack and dependency graphs to enhance situation awareness. In conclusion, our work differs from previous works in two major ways. First, we provide a mechanism to index alerts and recognize attacks in real-time. Second, we provide a mechanism to integrate attack and dependency graphs and enable real-time scenario analysis and better security decisions.

3 Generalized Dependency Graphs

Modern distributed systems typically consist of a large number of interdependent hardware and software components. Some dependencies may not even be explicit, and Leslie Lamport’s famous quote “*a distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable*” [14] greatly captures the essence of the problem. In the following, we will use the term *network entity*, or simply *entity*, to refer to any component of a distributed system at any level of abstraction (e.g., subnet, host, application, service). Dependencies amongst network entities can be broadly classified in the following three categories: *redundancy* (a network entity depends on a redundant pool of resources), *graceful degradation* (a network entity depends on a pool of entities; if one fails, the system can continue to work with degraded performance), and *strict dependence* (a network entity strictly depends on a pool of other entities; if one fails, the dependent entity becomes unusable).

In order to model all possible scenarios, we first assume that the status of each network entity can be represented as a performance indicator on a scale from 0 to 1, 1 meaning that the entity is fully working, and 0 meaning that the entity is unusable. We then assume the existence of a family $\mathcal{F}$ of functions of the form $f : [0, 1]^n \rightarrow [0, 1]$, which evaluate the performance of an entity given the performance of the n entities it depends on. We call these functions *dependency functions*, and require that $f(0, \dots, 0) = 0$ and $f(1, \dots, 1) = 1$.

The equations below define dependency functions corresponding to the three scenarios described above. If a dependency is described by f_r , then h will be fully operational if at least one of the entities it depends on is fully operational. If the dependency is described by f_s , all the entities need to be fully operational for h to be fully operational.

$$f_r(l_1, \dots, l_n) = \begin{cases} 1, & \text{if } \exists i \in [1, n] \text{ s.t. } l_i = 1 \\ 0, & \text{otherwise} \end{cases} \quad f_d(l_1, \dots, l_n) = \frac{1}{n} \sum_{i=1}^n l_i$$

$$f_s(l_1, \dots, l_n) = \begin{cases} 1, & \text{if } \forall i \in [1, n] l_i = 1 \\ 0, & \text{otherwise} \end{cases}$$

We can now introduce a generalization of dependency graphs that can capture not only which entities depend on which other entities, but also the nature of such dependencies. Service dependency models have been widely studied. In [13], Kheir *et al.* propose a service dependency representation that enables intrusion and response impact evaluation. However, although it is possible to automatically discover dependencies [3], the task of understanding the nature of such dependencies has not been fully automated yet. For the purpose of the proposed framework, we will assume that dependency graphs generated via existing tools are manually augmented by system administrators. Further research will be needed to fully automate this process.

Definition 1 (Generalized Dependency Graph). A generalized dependency graph is a labeled directed acyclic graph $D = (H, Q, \phi)$, where

- H is a set of nodes, corresponding to network entities;
- $Q = \{(h_1, h_2) \in H \times H \mid h_1 \text{ depends on } h_2\}$ is a set of edges;
- $\phi : H \rightarrow \mathcal{F}$ is a mapping that associates with each node $h \in H$ a function $f \in \mathcal{F}$, s.t. the arity of f is equal to the outdegree of h ¹.

For each node $h \in H$ we will use $h^\cdot$ to denote the set of entities that depend on h and h to denote the set of entities h depends on.

Figure 1(d) shows a dependency graph for the network of Figure 1(a). Both h_E and h_F strictly depend on h_G . Local database servers also depend on h_G , but they can continue to operate, with degraded performance, if h_G is compromised.

Without loss of generality, we assume an arbitrary but fixed time granularity, and use $\mathcal{T}$ to denote the set of all time points. We can now introduce the definition of *network status function*, which, for each time point t , maps a network entity to its performance at time t .

Definition 2 (Network Status Function). Given a generalized dependency graph $D = (H, Q, \phi)$, a network status function for D is a function $s : H \times \mathcal{T} \rightarrow [0, 1]$ such that $\forall h \in H$ and $\forall t \in \mathcal{T}$, $s(h, t) \leq f(s(h_{i_1}, t), \dots, s(h_{i_m}, t))$, where $f = \phi(h)$, and $h^\cdot = \{h_{i_1}, \dots, h_{i_m}\}$ is the set of entities h depends on.

¹ If h is a terminal node in the dependency graph (i.e., it does not depend on any other node), we assume $\phi(h)$ is the constant (0-ary) function 1.

The performance of an entity h is bounded by its corresponding dependency function, which represents a theoretical maximum. In practice, the performance may be lower if h has been directly compromised by an attack.

4 Attack Modeling

In this section, we start by recalling the definition of *attack graph* presented in [22], and then introduce the notion of *probabilistic temporal attack graph*. An attack graph is a directed graph having two types of vertices, exploits and security conditions. Exploits are host-bound vulnerabilities, whereas security conditions refer to the network states required or implied by exploits, such as privilege levels. In [22], an attack graph is formally defined as a directed graph $G = (V \cup C, R_r \cup R_i)$, where V is the set of known exploits, C is the set of relevant security conditions, and R_r and R_i denote the *require* and *imply* relationship between exploits and security conditions, defined as $R_r \subseteq C \times V$ and $R_i \subseteq V \times C$, respectively. The *prepare-for* relationship between exploits is the composite relation $R_i \circ R_r$. For the purpose of the type of analysis we present in this paper, we can assume, without loss of generality, that attack graphs are acyclic.

We extend the attack graph model discussed above with the notion of *timespan distribution*, which encodes probabilistic knowledge of the attacker's behavior as well as temporal constraints on the unfolding of attacks. We assume that each step of an attack sequence is completed within certain temporal windows after the previous exploit has been executed, each associated with a probability.

Example 1. Suppose an attacker has gained some privileges on host h_E in Figure 1. Using these privileges, he can then create the conditions to exploit a vulnerability on the main database server. However, this will take an amount of time depending on his skill level. The attacker will then have time to exploit the vulnerability until the vulnerability itself is patched, or the attack is discovered.

Leversage and Byres [15] describe how to estimate the *mean time to compromise* a system and relate that to the skill level of the attacker. This approach can be generalized to estimate timespan distributions for individual vulnerability exploits. In fact, we can assume that the time taken to exploit a vulnerability varies with the skill level of the attacker. Additionally, some vulnerabilities are easier to exploit than others, thus exhibiting higher probabilities. We will assume that timespan distributions are given, as a detailed discussion on how to derive them would be beyond the scope of this paper. The definitions of *timespan distribution* and *probabilistic temporal attack graph* are given below.

Definition 3 (Timespan Distribution). A timespan distribution $\omega \in \Omega$ is a pair (I, ρ) where:

- I is a set of time intervals² s.t. $\forall [x, y] \in I, x \leq y$;
- $\forall [x, y], [x', y'] \in I$ s.t. $[x, y] \neq [x', y']$, intervals $[x, y]$ and $[x', y']$ are disjoint;

² A time interval is a closed interval of the set $\mathcal{T}$ of all time points.

- $\rho : I \rightarrow [0, 1]$ is a function that associates a value $\rho(x, y) \in [0, 1]$ with each time interval $[x, y] \in I$.

Given a timespan distribution $\omega = (I, \rho)$, we use $S(\omega)$ to denote $\sum_{[x, y] \in I} \rho(x, y)$, and $\omega.t_{max}$ to denote $\max_{[x, y] \in I \mid \rho(x, y) > 0} y$, i.e., the maximum time point for which ρ is not 0. We require that $S(\omega) \leq 1$. $\square$

Intuitively, a timespan distribution (I, ρ) specifies a set I of disjoint time intervals when a given exploit might be executed, and an incomplete probability distribution ρ over I : $\rho(x, y)$ is the probability that the exploit will in fact be executed during the time interval $[x, y]$, following the execution of the previous exploit. ρ is not forced to be complete as the exploit might not be executed.

Definition 4 (Probabilistic Temporal Attack Graph). Given an attack graph $G = (V \cup C, R_r \cup R_i)$ a probabilistic temporal attack graph built on G is a labeled directed acyclic graph $A = (V, E, \delta, \gamma)$ where:

- V is the finite set of vulnerability exploits in the attack graph;
- $E = R_i \circ R_r$;
- $V^s = \{v \in V \mid \nexists v' \in V \text{ s.t. } (v', v) \in E\} \neq \emptyset$, and $V^e = \{v \in V \mid \nexists v' \in V \text{ s.t. } (v, v') \in E\} \neq \emptyset$, i.e., there exists at least one start node and one end node;
- $\delta : E \rightarrow \Omega$ is a function that associates a timespan distribution with each edge in the graph, such that $(\forall v \in V) \sum_{\{v' \in V \mid (v, v') \in E\}} S(\delta(v, v')) = 1$;
- γ is a function that associates with each exploit $v_j \in V \setminus V^s$ the condition

$$\gamma(v_j) = \bigwedge_{c_k \in C \text{ s.t. } (c_k, v_j) \in R_r} \left(\bigvee_{v_i \in V \text{ s.t. } (v_i, c_k) \in R_i} \nu_{i,j} \right),$$

where $\nu_{i,j}$ denotes that v_j must be preceded by v_i and must be executed within one of the time intervals defined by $\delta(v_i, v_j)$, following the execution of v_i . $\square$

Intuitively, an edge $e = (v_i, v_j)$ in a probabilistic temporal attack graph indicates that vulnerability exploit v_i prepares for exploit v_j . The timespan distribution $\delta(e) = (I, \rho)$ labeling the edge indicates how the probability that v_j is executed after v_i changes with time. The condition $\gamma(v_j)$ labeling a node v_j encodes the dependencies between v_j and the exploits preparing for it. In the original attack graph, each exploit v_j may require one or more security conditions to be satisfied, and each such condition may be achieved through several alternative exploits. This explains the conjunctive normal form of $\gamma(v_j)$. In the following, we will often abuse notation, and use v_i instead of $\nu_{i,j}$ in condition $\gamma(v_j)$, when v_j is clear from the context. Additionally, we say that a set V^* of exploits satisfies condition $\gamma(v_j)$ if exploits in V^* are sufficient to satisfy all the security conditions required by v_i .

Attacks can be executed in many different ways, which we will refer to as *instances*. The notion of attack instance is formalized in the following definition.

Definition 5 (Attack Instance). Given a probabilistic temporal attack graph $A = (V, E, \delta, \gamma)$, an instance of A is a tree $T = (V_T, E_T)$ over A , rooted at an end node of A and defined as follows:

- $|V_T \cap V^e| = 1$, i.e., there is exactly one end node of A in T ;
- $(\forall v \in V_T \setminus V^s) \exists V' \subset V_T$ s.t. V' satisfies $\gamma(v) \wedge \nexists V'' \subset V'$ s.t. V'' satisfies $\gamma(v)$. $\square$

An instance is represented as a tree because each attack pattern aims at creating a certain target condition (e.g., gaining access to the database server of Figure 1(a)), which can be achieved by executing an exploit corresponding to an end node in the temporal attack graph (the root of the tree). Each exploit may require one or more exploits being executed in preparation for it, and the leaves of the tree represent exploits which depend on initial security conditions.

5 Combining Attack and Dependency Graphs

Dependency and attack graphs are great tools to analyze raw security data and derive useful insights on the structure and weaknesses of a network. The following example clearly shows that neither of these tools, if used alone, can provide a cyber security analyst with enough information to make good decisions. Instead, we need a tool that can combine information from the two graphs, also enabling the analyst to import his risk analysis knowledge.

Example 2. Consider the network of Figure 1(a), and suppose an attacker has compromised host h_C . Based on the attack graph of Figure 2(a), an automated system may recommend the analyst to protect host h_D , as a vulnerability on h_D is the most likely to be exploited next. However, when we also consider information in the dependency graph of Figure 1(d), this clearly turns to be the worst choice possible. In fact, the only entity which directly depends on h_D is host h_C , which is already compromised. Thus, the marginal damage caused by the attacker exploiting h_D is negligible. Conversely, exploiting the vulnerability on host h_F , although less likely, would cause a much larger damage because it would also disrupt the Web Server and the Online Shopping service.

The key observation behind our approach to combining attack and dependency graphs is that the execution of a vulnerability (i.e., a node in the attack graph) might cause a more or less severe reduction in performance in one or more network entities (nodes in the dependency graph). This, in turn, may affect other entities not directly affected by the exploit. We can thus introduce the following fundamental definition.

Definition 6 (Attack Scenario Graph). *Given a probabilistic temporal attack graph $A = (V, E, \delta, \gamma)$ and a generalized dependency graph $D = (H, Q, \phi)$, an attack scenario graph is a 4-tuple (A, D, F, η) where*

- $F \subseteq V \times H$;
- $\eta : F \rightarrow [0, 1]$ is a function that associates with each pair $(v, h) \in F$ a real number in the $[0, 1]$ interval representing the percentage reduction in the performance of entity h caused by vulnerability exploit v . $\square$

makes h_C totally unusable (as $\eta(v_C, h_C) = 1$), leading to $s_1(h_C) = 0$. As h_T strictly depends on h_C , we also have $s_1(h_T) = 0$. Therefore, the marginal damage of exploit v_C is $\Delta damage_1 = 14$. At time t_2 , the attacker may take one of two possible steps: exploiting v_D (with probability 0.7) or v_F (with probability 0.3). In the first case, $\Delta damage_2 = 0.7 \cdot 5 = 3.5$ (h_C and h_T are already unusable because of the previous exploit). In the second case, $\Delta damage_2 = 0.7 \cdot 7 + 8 + 10 = 22.9$ (h_F is partially compromised, making h_A and h_S unusable). In conclusion, the second alternative, although less likely, can cause a much larger damage.

When attack and dependency graphs are very large (millions of nodes), portions of them could be maintained on disk rather than in main memory. In this case, a merged data structure would provide additional advantages over two separate graphs. If nodes from the two graphs pertaining the same group of machines are stored in the same block on disk, all the information needed to process alerts from that set of machines could be loaded with a single access to disk.

6 Real-time Scenario Analysis

In this section, we formalize the problem of real-time scenario analysis, and propose a data structure to monitor and index incoming alerts, as well as an algorithm to update the index in real-time. For the purpose of our analysis, we assume that each alert o is a tuple $(type, ts, h_{src}, h_{dest})$, where $type$ denotes the event type, $ts \in T$ is a timestamp, and h_{src}, h_{dest} are the source and destination host respectively. We refer to a sequence O of such alerts as the *observation sequence*. Finally, we assume the existence of a function $exploit : O \rightarrow V$ that maps alerts in O to vulnerability exploits. Informally, we define an *occurrence* of an attack as a sequence of alerts constituting the evidence that a given attack occurred. A formal definition is given below.

Definition 7 (Attack Occurrence). *Given a probabilistic temporal attack graph $A = (V, E, \delta, \gamma)$, and an observation sequence O , an occurrence of A in O with probability p is a sequence $O^* = \langle o_1^*, \dots, o_k^* \rangle \subseteq O$ such that:*

- $o_1^*.ts \leq o_2^*.ts \leq \dots \leq o_k^*.ts$;
- *there exists an instance $T = (V_T, E_T)$ of A s.t. $(\forall (v_i, v_j) \in E_T) \exists o_i, o_j \in O^*$ s.t. $exploit(o_i) = v_i \wedge exploit(o_j) = v_j \wedge o_i.ts \leq o_j.ts$, i.e., O^* includes alerts corresponding to all the exploits in T ;*
- $prob(\text{root}(V_T)) \geq p$, where $prob$ is recursively defined as

$$(\forall v \in V_T) prob(v) = \begin{cases} 1, & \text{if } v \text{ is a leaf node} \\ \prod_{v' \in \text{children}(v)} \rho(x, y) \cdot prob(v') & \end{cases}$$

where $(I, \rho) = \delta(v', v)$, $[x, y] \in I$ s.t. $x \leq o.ts - o'.ts \leq y$, and $exploit(o) = v$, $exploit(o') = v'$.

The span of the occurrence above as the time interval $\text{span}(O^*) = [o_{i_1}.ts, o_{i_k}.ts]$. $\square$

In other words, an occurrence of a probabilistic temporal attack A is a sequence of alerts indicating that all the exploits in an instance of the temporal attack graph have been executed in the right order and satisfying all the temporal constraints, thus enabling the attacker to achieve his target security condition.

We are interested in recognizing occurrences of attacks in real-time. Note that multiple concurrent attacks generate interleaved alerts in the observation sequence. In general, the problem of finding all possible occurrences of attacks is exponential in the level of concurrency (i.e., the maximum number of concurrent attacks), and quadratic in the number of received alerts. However, we will show that, leveraging temporal features of attack graphs, it is possible to compute all the valid solutions in time linear to the number of received alerts.

In order to concurrently monitor multiple types of attacks, we first merge all probabilistic temporal attack graphs from a set $\mathcal{A} = \{A_1, \dots, A_k\}$ – with $A_i = (V_i, E_i, \delta_i, \gamma_i)$ and $I_{\mathcal{A}} = \{id(A_1), \dots, id(A_k)\}$ – into a single *multi-attack graph* $M = (V_M, I_{\mathcal{A}}, \delta_M, \gamma_M)$ where (i) $V_M = \cup_{i=1}^k V_i$ is a set of vulnerability exploits; (ii) $\delta_M : V_M \times V_M \times I_{\mathcal{A}} \rightarrow \Omega$ is a function that associates a triple $(v, v', id(A_i))$ with $\delta_i(v, v')$, if $(v, v') \in E_i$, or **null** otherwise; (iii) γ_M is a function that associates each pair $(v, id(A_i)) \in V_M \times I_{\mathcal{A}}$ with $\gamma_i(v)$, if $v \in V_i \setminus V_i^s$, or **null** otherwise. In the following, we will assume that an attack scenario graph (Definition 6) is a 4-tuple (M, D, F, η) , where M is a multi-attack graph, rather than a single attack graph. We can now give the definition of *attack scenario index*, a data structure that enables indexing and tracking of ongoing attacks, and provides support for analysis of future scenarios.

Definition 8 (Attack Scenario Index). *Let $G = (M, D, F, \eta)$ be an attack scenario graph, where $M = (V_M, I_{\mathcal{A}}, \delta_M, \gamma_M)$ is a multi-attack graph built over a set $\mathcal{A} = \{A_1, \dots, A_k\}$ of probabilistic temporal attack graphs, and $D = (H, Q, \phi)$ is a generalized dependency graph. An attack scenario index is a 6-tuple $I_G = (G, start_G, end_G, tables_G, s_G, completed_G)$, where:*

- $start_G$ (resp. end_G) : $V_M \rightarrow 2^{I_{\mathcal{A}}}$ is a function that associates with each node $v \in V_M$, the set of attack graph id's for which v is a start (resp. end) node;
- For each $v \in V_M$, $tables_G(v)$ is a set of records of the form $(curr, attackID, ts_0, prob, \Delta damage, prev, next)$, where $curr$ is a reference to an alert, $attackID \in I_{\mathcal{A}}$ is an attack graph id, $ts_0 \in T$ is a timestamp, $prob \in [0, 1]$ is the probability of the (partial) occurrence, $\Delta damage \in \mathbb{R}$ is the marginal damage, $prev$ and $next$ are sets of references to records in $tables_G$;
- $s_G : H \rightarrow [0, 1]$ is the current status of the network;
- $completed_G : I_{\mathcal{A}} \rightarrow 2^{\mathcal{P}}$, where $\mathcal{P}$ is the set of references to records in $tables_G$, is a function that associates with each attack identifier $id(A)$ a set of references to records in $tables_G$ corresponding to completed occurrences of A . $\square$

Note that G , $start_G$, end_G can be computed a-priori, given the set $\mathcal{A}$ of attack graphs and the dependency graph D . All the tables that are part of the index ($tables_G$) will be initially empty. As new alerts are received, they will be updated accordingly, as described in Section 6.1. Index tables allow to track partially completed occurrences, and each record points to an alert, as well as to previous

Algorithm 1. $updateIndex(o_{new}, I_G, p_t)$ **Input:** New alert to be processed o_{new} , attack scenario index I_G , probability threshold p_t .**Output:** Updated attack scenario index I_G .

```

1:  $v_{new} \leftarrow exploit(o_{new})$  // Map the new alert to a known vulnerability exploit
2:  $\Delta damage \leftarrow 0$ 
3: for all  $h \in H$  s.t.  $(v_{new}, h) \in F$  do
4:    $\Delta damage \leftarrow \Delta damage + assessDamage(h, \eta(v_{new}, h))$ 
5: // Look at start nodes
6: if  $start_G(v_{new}) \neq \emptyset$  then
7:   for all  $id \in start_G(v_{new})$  do
8:     add  $(o_{new}, id, o_{new}.ts, 1, \Delta damage, \emptyset, \emptyset)$  to  $tables_G(v_{new})$ 
9: // Look at intermediate nodes
10:  $V \leftarrow \emptyset$ 
11: for all node  $v \in V_M$  s.t.  $\exists id \in I_{\mathcal{A}}, \delta_M(v, v_{new}, id) \neq \text{null}$  do
12:    $r_{first} \leftarrow \min\{r \in tables_G(v) \mid o_{new}.ts - r.curr.ts \leq \max_{id \in I_{\mathcal{A}} \mid \delta_M(v, v_{new}, id) \neq \text{null}} \delta_M(v, v_{new}, id).tmax\}$ 
13:   for all records  $r \in tables_G(v)$  s.t.  $r \geq r_{first}$  do
14:      $id \leftarrow r.attackID$ 
15:     if  $\delta_M(v, v_{new}, id) \neq \emptyset$  then
16:        $(I, \tau) \leftarrow \delta_M(v, t_{new}.obs, id)$ 
17:        $p \leftarrow \tau(x, y)$  where  $[x, y] \in I$  and  $x \leq t_{new}.ts - r.curr.ts \leq y$ 
18:       if  $p \geq p_t$  then
19:          $V \leftarrow V \cup \{(v, r, p)\}$ 
20:         if  $V$  satisfies  $\gamma(v_{new})$  then
21:            $prob \leftarrow 1$ 
22:            $records \leftarrow \emptyset$ 
23:           for all  $(v_i, p_i, r_i) \in V^*$  s.t.  $V^*$  is a minimal subset of  $V$  satisfying  $\gamma(v_{new})$  do
24:              $prob \leftarrow prob \cdot r_i.prob \cdot p_i$ 
25:              $records \leftarrow records \cup \{r_i^\uparrow\}$ 
26:           if  $prob \geq p_t$  then
27:              $r_n \leftarrow (o_{new}, id, \min_{r' \in records} r'.ts_0, prob, \Delta damage, records, \emptyset)$ 
28:             add  $r_n$  to  $tables_G(v_{new})$ 
29:             for all  $r' \in records$  do
30:                $r'.next \leftarrow r'.next \cup \{r_n^\uparrow\}$ 
31:             // Look at end nodes
32:             if  $id \in end_G(v_{new})$  then
33:               add  $r_n^\uparrow$  to  $completed_G(id)$ 

```

and successor records. Each record also stores the time at which the partial occurrence began (ts_0), and the marginal damage caused by the corresponding exploit.

6.1 Index Update Algorithm

This section describes an algorithm (Algorithm 1, $updateIndex$) to update the index when a new alert is received. The algorithm takes as input an attack scenario index I_G , a new alert o_{new} , and a probability threshold p_t .

Line 1 maps the newly received alert o_{new} to a known vulnerability exploit v_{new} . Lines 2–4 compute the marginal damage caused by exploit v_{new} by invoking the $assessDamage$ algorithm (Algorithm 2) for each network entity $h \in H$ directly affected by v_{new} and summing up all the contributions. Algorithm 2 starts by computing the reduction in the performance of h (Lines 1–5), and then iteratively propagates damage assessment to all the affected nodes (Lines 7–8).

Lines 6–8 of Algorithm 1 handle the case when v_{new} is the start node of an attack graph in $\mathcal{A}$. A new record is added to $start_G(v_{new})$ for each attack graph in $\mathcal{A}$ for which v_{new} is a start node. Lines 10–33 look at the tables associated with the nodes that precede v_{new} in the temporal multi-attack graph

Algorithm 2. *assessDamage*(h, η)

Input: Node h in the generalized dependency graph, performance reduction η due to an exploit
Output: marginal damage assessment

```

1: // Update status and assess direct damage to  $h$ 
2:  $s_{before} \leftarrow s_G(h)$ 
3:  $s_G(h) \leftarrow \min((1 - \eta) \cdot s_G(h), \phi(h)(h))$ 
4:  $s_{after} \leftarrow s_G(h)$ 
5:  $\Delta_{damage} \leftarrow (s_{before} - s_{after}) \cdot u(h)$ 
6: // Assess damage propagation
7: for all  $h_i \in h$  s.t.  $s_G(h_i) > 0$  do
8:    $\Delta_{damage} \leftarrow \Delta_{damage} + \text{assessDamage}(h_i, 0)$ 
9: return  $\Delta_{damage}$ 

```

and check whether the new alert can be correlated to existing partially completed occurrences. For each predecessor v of v_{new} , Line 12 determines where the algorithm should start scanning $tables_G(v)$. Note that records are added to index tables as new alerts are received, thus they are ordered by $r.curr.ts$, i.e., the time at which the corresponding alert was received. Given two records $r_1, r_2 \in tables_G(v)$, we use $r_1 \leq r_2$ to denote the fact that r_1 precedes r_2 in $tables_G(v)$, i.e., $r_1.curr.ts \leq r_2.curr.ts$. To avoid scanning the entire table, only the “most recent” records in $tables_G(v)$ are considered, i.e., those that can still be linked to new alerts (Line 12). On Lines 16-17, timespan distributions are used to determine the probability p that the new alert can be linked to record r for the attack graph identified by $r.attackID$, given the amount of time elapsed between $r.curr.ts$ and $o_{new}.ts$. Since we are interested in occurrences with a probability above the threshold p_t , we discard the partial occurrence as soon as $p < p_t$ (Line 18). Otherwise, we keep track in V of all the predecessor exploits that can be linked to v_{new} , and, if the condition $\gamma(v_{new})$ is satisfied (Line 20), we consider a subset V^* of V that minimally satisfies $\gamma(v_{new})$ and add a new record r_n to $tables_G(v)$ (Lines 27-28) if the overall probability of the corresponding (partial) occurrence is above the threshold. Additionally, we update predecessor records to have r_n as their successor (Lines 29-30). Note that r_n inherits ts_0 from its predecessors; this ensures that the starting and ending times can be quickly retrieved by looking directly at the last record for a completed occurrence. Finally, lines 32–33 check whether v_{new} is an end node for some attack graph. If yes, a pointer to r_n is added to $completed_G$, telling that a new occurrence has completed.

Algorithm *updateIndex*, can be used iteratively for loading an entire observation sequence at once (we refer to this variant as *bulkUpdate*). The following result characterizes the time complexity of the two algorithms.

Proposition 1. *Given a set $\mathcal{A}$ of probabilistic temporal attack graphs, a multi-attack graph $M = (V_M, I_A, \delta_M, \gamma_M)$ over $\mathcal{A}$, an attack scenario graph G , and an attack scenario index $I_G = (G, start_G, end_G, tables_G, s_G, completed_G)$, the worst case complexity of algorithm *updateIndex* (resp. *bulkUpdate*) is $O(k^{|V_M|} \cdot |\mathcal{A}|)$ (resp. $O(k^{|V_M|} \cdot |\mathcal{A}| \cdot |O|)$), where O is the observation sequence and k is the level of concurrency, i.e., the maximum number of concurrent attacks. $\square$*

Note that the complexity result above are based on the fact that the number of *recent* records in each table $tables_G(v)$ is independent of the size of O . This can be formally proved, and is largely confirmed by experimental results. Additionally, our experiments show that, in practice, time and space complexity are independent of the size of the activities. This result is expected since $|O| \gg |V_M|$.

7 Analysis of Future Scenarios

As Example 3 has shown, recommendations should be made to the analyst based on a compromise between likelihood and marginal damage of possible future scenarios. Ideally, an automated system should provide the analyst with a list of predicted scenarios, ranked by a measure of criticality accounting for both probability and marginal damage. The analyst would then be recommended to take action to prevent attack steps corresponding to the scenario with the highest criticality. In the simplest case, given a set of predicted occurrences $\{O_1^*, \dots, O_n^*\}$, we can estimate the criticality of O_i^* , for each $i \in [1, n]$, as $p_i \cdot \Delta damage_i$, where p_i and $\Delta damage_i$ are the probability and marginal damage of O_i^* respectively. In general, a *criticality function* can be defined as any function of the form $f : [0, 1] \times \mathbb{R} \rightarrow \mathbb{R}$ that satisfies the following monotonicity axioms:

$$(\forall \Delta d \in R) \ p_1 \geq p_2 \Rightarrow f(p_1, \Delta d) \geq f(p_2, \Delta d) \quad (3)$$

$$(\forall p \in [0, 1]) \ \Delta d_1 \geq \Delta d_2 \Rightarrow f(p, \Delta d_1) \geq f(p, \Delta d_2) \quad (4)$$

Example 4. Consider the two scenarios described in Example 3. According to the simple computation model described above, their criticality is 2.45 and 6.87 respectively, confirming that protecting h_F would be a wiser choice.

We propose algorithm *rankFutureScenarios* (Algorithm 3) to predict and rank all possible future scenarios. We assume that the attacker's goal is to reach a network resource corresponding to the end node of an attack graph. The algorithm takes as input an attack scenario graph I_G , a time point ts , an integer $k > 0$, a probability threshold p_t , and a criticality function f , and returns a ranked list of possible future attack occurrences. The algorithm considers all the record in the index which do not have a successor yet (Lines 2-3). For each such record r , a recursive algorithm (*simGraphForward*) traverses the attack graph forward, and derives a set of partial future occurrences of length k or less along with their probability and marginal damage assessment (Line 5). This algorithm is straightforward and details are omitted for reasons of space. Finally, all the predicted partial occurrences are ranked using f . The following result characterizes the time complexity of algorithms *rankFutureScenarios*.

Proposition 2. *Given an attack scenario index $I_G = (G, start_G, end_G, tables_G, s_G, completed_G)$ and an integer k , the worst case complexity of algorithm *rankFutureScenarios* is $O(|V_M| \cdot d^k)$, where V_M is the set of exploits in the multi-attack graph and d is the maximum out-degree of nodes in V_M . $\square$*

Algorithm 3. *rankFutureScenarios*(I_G, ts, k, p_t, f)**Input:** Attack scenario index I_G , time ts , integer $k > 0$, prob. threshold p_t , criticality function f .**Output:** List of 4-tuples $(id(A), O, prob, \Delta damage)$ ranked by $f(prob, \Delta damage)$, where $id(A) \in I_A$, O is a partial occurrence, $prob$ is the probability of O , and $\Delta damage$ is the marginal damage

```

1:  $S \leftarrow \emptyset$ 
2: for all  $v \in V_M$  do
3:   for all  $r \in tables_G(v)$  s.t.  $e.next = \emptyset \wedge r.curr.ts < ts \wedge r^\dagger \notin completed_G(r.attackID)$  do
4:     // Find attack patterns starting at  $v$ , no longer than  $k$ , and with probability above  $p_t$ 
5:      $O \leftarrow simGraphForward(r.attackID, exploit(r.curr), k, p_t)$ 
6:     for all  $(O, prob, \Delta damage) \in \mathcal{O}$  do
7:        $S \leftarrow S \cup \{(r.attackID, O, prob, \Delta damage)\}$ 
8: return rankedList( $S, f$ )

```

The above result, confirmed by the experiments, indicates that, although time complexity is exponential in the number k of steps forward, the effect of the exponential is not significant. In fact, typically $|V_M| \gg d$, and k is not large.

8 Experiments

In this section, we report the results of the experiments we conducted to evaluate the time and memory performance of the proposed index update algorithm, as well as the performance of the *rankFutureScenario* algorithm. We evaluated the system using both real and synthetic attack graphs. In both cases, we used the attack graphs to simulate a number of attacks and generate a stream of alerts. Specifically, we used an attack graph that was generated by scanning an existing network with the tool described in [11]. The resulting temporal attack graph includes 786 nodes, encompassing 64 machines. In order to test our framework on larger graphs, we generated synthetic graphs of up to 300 thousand nodes.

Figure 3(a) shows how the time to build the entire index using the *bulkUpdate* algorithm – for both synthetic and real attack graphs – increases as the number of alerts increases. It is clear that the index building time is linear in the number of alerts (note that both axes are on a log scale), and the algorithm can process between 25 and 30 thousands alerts per second. Also note that there is no significant difference between results on real and synthetic attack graphs, and that the size of the graphs does not significantly affects the index building time, as confirmed by Figure 3(b): when the size of the merged graph changes by orders of magnitude, the processing time increases slightly, but remains within the same order of magnitude. This can be easily explained by considering that, for a given number of alerts, when the size of the graphs increases, the number of index tables (one for each $v \in V_M$) increases as well, but at the same time the average number of occurrences of each alert and the average size of each table decrease, keeping total memory occupancy (Figure 3(c)) and processing time (Figure 3(b)) almost constant. The slight increase in processing time can then be attributed to the overhead of managing a larger number of tables. Figure 3(c) also indicates that memory occupancy increases linearly with the number of alerts processed.

Finally, execution times of algorithm *rankFutureScenarios* for varying values of k are reported in Figure 3(d). Note that the processing time is not

Fig. 3. Experimental results

significantly affected by the size of the graphs. This can be easily explained with the same argument used above for index building time and memory. Algorithm *rankFutureScenarios* looks at the most recent records in each index table, i.e., those in a given temporal window. As the size of the graphs increases, the number of records per table falling in such temporal window decreases, thus keeping the total number of records to examine almost constant. For any given graph size, when k increases, the processing time increases. However, we can note that, for smaller graphs, the processing time becomes stable as k increases, whereas, for larger graphs, it will continue to increase exponentially. This can be easily explained considering that, as k increases, it will be less likely for smaller graphs to have partial patterns of length k , so in most cases the algorithm will end the recursive search earlier than k steps forward.

9 Conclusions

In this paper, we proposed a novel framework to integrate vulnerability and dependency analysis, and provide security analysts with a better picture of the cyber situation. Our contribution was threefold. First, we introduced the notion of generalized dependency graph, which captures how network components, at different levels of abstraction, depend on each other. Second, we extended the classical definition of attack graph to incorporate probabilistic knowledge of the attacker's behavior. Finally, we introduced the notion of attack scenario graph,

which integrates dependency and attack graphs. We proposed efficient algorithms for detection and prediction, and showed that our framework can handle very large attack graphs and large volumes of alerts. Further research will be needed to fully automate the generation of attack scenario graphs.

References

1. Albanese, M., Chellappa, R., Moscato, V., Picariello, A., Subrahmanian, V.S., Turaga, P., Udea, O.: A constrained probabilistic petri net framework for human activity detection in video. *IEEE Transactions on Multimedia* 10(8), 1429–1443 (2008)
2. Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D.A., Zhang, M.: Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review* 37, 13–24 (2007)
3. Bahl, P.V., Barham, P., Black, R., Chandra, R., Goldszmidt, M., Isaacs, R., Kandula, S., Li, L., MacCormick, J., Maltz, D., Mortier, R., Wawrzoniak, M., Zhang, M.: Discovering Dependencies for Network Management. In: *Proceedings of the 5th ACM Workshop on Hot Topics in Networking (HotNets)* (November 2006)
4. Chen, X., Zhang, M., Mao, Z.M., Bahl, P.: Automating network application dependency discovery: experiences, limitations, and new solutions. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008*, pp. 117–130. USENIX Association, Berkeley (2008)
5. Dain, O., Cunningham, R.K.: Fusing a heterogeneous alert stream into scenarios. In: *Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications*, pp. 1–13 (2001)
6. Duong, T., Bui, H., Phung, D., Venkatesh, S.: Activity Recognition and Abnormality Detection with the Switching Hidden Semi-Markov Model. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, vol. 1, pp. 838–845 (2005)
7. Golab, L., Özsu, M.T.: Issues in data stream management. *SIGMOD Record* 32, 5–14 (2003)
8. Habra, N., Charlier, B., Mounji, A., Mathieu, I.: Asax: Software architecture and rule-based language for universal audit trail analysis. In: Deswarte, Y., Eizenberg, G., Quisquater, J.-J. (eds.) *ESORICS 1992*. LNCS, vol. 648, pp. 435–450. Springer, Heidelberg (1992)
9. Hamid, R., Huang, Y., Essa, I.: ARGMode Activity Recognition Using Graphical Models. In: *Proceedings of the IEEE Computer Society International Conference on Computer Vision and Pattern Recognition (CVPR 2003)*, vol. 3, pp. 38–43 (2003)
10. Ingols, K., Lippmann, R., Piwowarski, K.: Practical attack graph generation for network defense. In: *Proceedings of 22nd Annual Computer Security Applications Conference (ACSAC 2006)*, pp. 121–130. IEEE Computer Society, Los Alamitos (2006)
11. Jajodia, S., Noel, S.: Topological Vulnerability Analysis. In: *Cyber Situational Awareness: Issues and Research*. Advances in Information Security, vol. 46, pp. 139–154. Springer, Heidelberg (2009)
12. Kandula, S., Chandra, R., Katabi, D.: What’s going on?: learning communication rules in edge networks. *ACM SIGCOMM Computer Communication Review* 38, 87–98 (2008)

13. Kheir, N., Cuppens-Boulahia, N., Cuppens, F., Debar, H.: A service dependency model for cost-sensitive intrusion response. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 626–642. Springer, Heidelberg (2010)
14. Lamport, L.: Distributed system (May 1987), <http://research.microsoft.com/enus/um/people/lamport/pubs/distributed-system.txt>
15. Leversage, D.J., Byres, E.J.: Estimating a system's mean time-to-compromise. IEEE Security and Privacy 6, 52–60 (2008)
16. Mörchén, F.: Unsupervised pattern mining from symbolic temporal data. SIGKDD Explorations Newsletter 9(1), 41–55 (2007)
17. Ning, P., Xu, D.: Learning attack strategies from intrusion alerts. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003), pp. 200–209 (2003)
18. Ning, P., Xu, D., Healey, C.G., Amant, R.S.: Building attack scenarios through integration of complementary alert correlation methods. In: Proceedings of the 11th Annual Network and Distributed System Symposium (NDSS 2004), pp. 97–111 (2004)
19. Noel, S., Robertson, E., Jajodia, S.: Correlating intrusion events and building attack scenarios through attack graph distances. In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004), pp. 350–359 (2004)
20. Qin, X., Lee, W.: Statistical causality analysis of INFOSEC alert data. In: Vigna, G., Krügel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 73–93. Springer, Heidelberg (2003)
21. Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 54–68. Springer, Heidelberg (2001)
22. Wang, L., Liu, A., Jajodia, S.: Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts. Computer Communications 29(15), 2917–2933 (2006)