# Scalable Detection of Cyber Attacks[⋆]

Massimiliano Albanese[1], Sushil Jajodia[1], Andrea Pugliese[2], and V.S. Subrahmanian[3]

[1] George Mason University, Fairfax, VA 22030, USA
{malbanes,jajodia}@gmu.edu
[2] University of Calabria, 87036 Rende (CS), Italy
apugliese@deis.unical.it
[3] University of Maryland, College Park, MD 20742, USA
vs@umiacs.umd.edu

**Abstract.** Attackers can exploit vulnerabilities to incrementally penetrate a network and compromise critical systems. The enormous amount of raw security data available to analysts and the complex interdependencies among vulnerabilities make manual analysis extremely labor-intensive and error-prone. To address this important problem, we build on previous work on topological vulnerability analysis, and propose an automated framework to manage very large attack graphs and monitor high volumes of incoming alerts for the occurrence of known attack patterns in real-time. Specifically, we propose (i) a data structure that merges multiple attack graphs and enables concurrent monitoring of multiple types of attacks; (ii) an index structure that can effectively index millions of time-stamped alerts; (iii) a real-time algorithm that can process a continuous stream of alerts, update the index, and detect attack occurrences. We show that the proposed solution significantly improves the state of the art in cyber attack detection, enabling real-time attack detection.

**Keywords:** Attack graphs, attack detection, scalability

## 1 Introduction

An ever increasing number of critical applications and services rely today on Information Technology infrastructures, exposing companies and organizations to an elevated risk of becoming the target of cyber attacks. Attackers can exploit network configurations and vulnerabilities to incrementally penetrate a network and compromise critical systems. Most of the elementary steps of an attack are intercepted by intrusion detection systems, which generate alerts accordingly. However, such systems typically miss some events and also generate a large number of false alarms. More importantly, they cannot derive attack scenarios from individual alerts.

[⋆] This material is based upon work supported by the Army Research Office under MURI grant W911NF-09-1-0525 and DURIP grant W911NF-11-1-0340.

The enormous amount of raw security data available to analysts, and the complex interdependencies among vulnerabilities make manual analysis extremely labor-intensive and error-prone. Although significant progress has been made in several areas of computer and network security, powerful tools capable of making sense of this ocean of data and providing analysts with the "big picture" of the cyber situation in a scalable and effective fashion are still to be devised. To address this important problem, we build on previous work on topological vulnerability analysis [7, 14], and propose an automated framework to manage very large attack graphs and monitor high volumes of incoming alerts for the occurrence of known attack patterns in real-time.

Specifically, we propose (i) a data structure that merges multiple attack graphs and enables concurrent monitoring of multiple types of attacks; (ii) an index structure that can effectively index millions of time-stamped alerts; (iii) a real-time algorithm that can process a continuous stream of alerts and update the index. We show that the proposed solution significantly improves the state of the art in real-time attack detection. Previous efforts have in fact indicated that it is possible to process alerts fast, under certain circumstances, but have not considered the impact of very large attack graphs. Finally, we report on preliminary experiments, and show that the proposed solution scales well for large graphs and large amounts of security alerts.

The paper is organized as follows. Section 2 discusses related work. Section 3 introduces the notion of *temporal attack graph*, whereas Section 4 presents the proposed data structures and the algorithm to process and index security alerts. Finally, Section 5 reports the results of experiments, and Section 6 provides some concluding remarks.

## 2 Related Work

To reconstruct attack scenarios from isolated alerts, some correlation techniques employ prior knowledge about attack strategies [3] or alert dependencies [9]. Some techniques aggregate alerts with similar attributes [13] or statistical patterns [12]. Hybrid approaches combine different techniques for better results [9]. To the best of our knowledge, the limitation of the nested-loop approach, especially for correlating intensive alerts in high-speed networks, has not been addressed yet. Network vulnerability analysis enumerates potential attack sequences between fixed initial conditions and attack goals [7]. In [10], Noel *et al.* adopt a vulnerability-centric approach to alert correlation, because it can effectively filter out bogus alerts irrelevant to the network. However, the nested loop procedure is still used in [10]. Real-Time detection of isolated alerts is studied in [11]. Designed for a different purpose, the RUSSEL language is similar to our approach in that the analysis of data only requires one-pass of processing [5].

Hidden Markov Models (HMMs) and their variants have been used extensively for plan recognition. Luhr et al. [8] use Hierarchical HMMs to learn probabilities of sequences of activities. [4] introduces the Switching Hidden Semi-Markov Model (SHSMM), a two-layered extension of the Hidden Semi-Markov

Model (HSMM). The bottom layer represents atomic activities and their duration using HSMMs, whereas the top layer represents a sequence of high-level activities defined in terms of atomic activities. In [6], Hamid et al. assume that the structure of relevant activities is not fully known a priori, and provide minimally supervised methods to learn activities. They make the simplifying assumption that activities do not overlap. The problem of recognizing multiple interleaved activities has been studied in [2], where the authors propose a symbolic plan recognition approach, relying on hierarchical plan-based representation and a set of algorithms that can answer a variety of recognition queries.

To the best of our knowledge, there has been virtually no work on efficient indexing to support scalable and real-time attack detection. Our work differs from previous work by providing a mechanism to index alerts as they occur in a data structure that also unifies a set of known attack graphs. The index we propose in this paper extends the index proposed by Albanese et al. [1]. Differently from [1], the index we propose in this paper can efficiently manage both large attack graphs and large sequences of alerts, whereas the authors of [1] do not address scale issues with respect to the size of the graphs.

## 3 Temporal Attack Graphs

In this paper, we extend the attack graph model of [14] with temporal constraints on the unfolding of attacks. We assume that each step of an attack sequence is taken within a certain temporal window after the previous step has been taken. Without loss of generality, we assume an arbitrary but fixed time granularity. We use $\mathcal{T}$ to denote the set of all time points. The definition of *temporal attack graph* is given below.

**Definition 1 (Temporal Attack Graph).** *Given an attack graph $G = (V \cup C, R_r \cup R_i)$ a* 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;*
- *$E = R_i \circ R_r$ is the* prepare for *relationship between exploits;*
- *$V^s = \{v \in V \mid \nexists v' \in V \ s.t. \ (v', v) \in E\} \neq \emptyset$, i.e., there exists at least one* start node *in $V$;*
- *$V^e = \{v \in V \mid \nexists v' \in V \ s.t. \ (v, v') \in E\} \neq \emptyset$, i.e., there exists at least one* end node *in $V$;*
- *$\delta : E \rightarrow \mathcal{T} \times \mathcal{T}$ is a function that associates a pair $(t_{min}, t_{max})$ with each edge in the graph;*
- *$\gamma$ is a function that associates with each vulnerability $v_i \in V \setminus V^s$ the condition*

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

*where each $\nu_{j,i}$ denotes that $v_i$ must be execute after $v_j$, within the temporal window defined by $\delta(v_j, v_i)$.* □

Intuitively, an edge $e = (v_j, v_i)$ in a temporal attack graph denotes the fact that exploit $v_j$ *prepares for* exploit $v_i$. The pair $\delta(e) = (t_{min}, t_{max})$ labeling the edge indicates that the time elapsed between the two exploits must be in the interval $[t_{min}, t_{max})$. The condition $\gamma(v_i)$ labeling a node $v_i$ encodes the dependencies between $v_i$ and all the exploits preparing for it. In the following, we often abuse notation and use $v_j$ instead of $\nu_{j,i}$ in condition $\gamma(v_i)$, when $v_i$ is clear from the context. We say that a set $V^*$ of exploits satisfies condition $\gamma(v_i)$ if exploits in $V^*$ imply all the security conditions *required* by $v_i$.

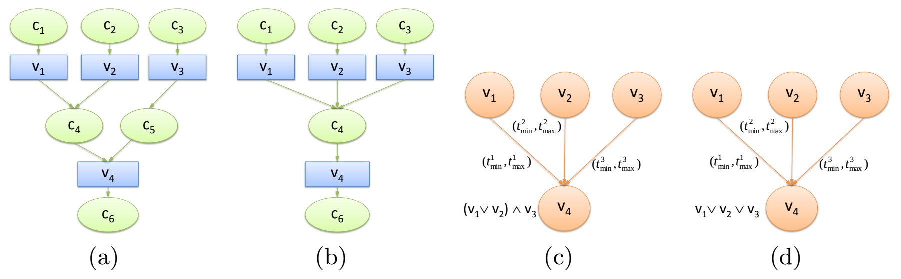

c1 c2 c3
v1 v2 v3
c4 c5
v4
c6
(a)
c1 c2 c3
v1 v2 v3
c4
v4
c6
(b)
v1 v2 v3
$(t^1_{min}, t^1_{max})$ $(t^2_{min}, t^2_{max})$ $(t^3_{min}, t^3_{max})$
$(v_1 \lor v_2) \land v_3$ v4
(c)
v1 v2 v3
$(t^1_{min}, t^1_{max})$ $(t^2_{min}, t^2_{max})$ $(t^3_{min}, t^3_{max})$
$v_1 \lor v_2 \lor v_3$ v4
(d)

**Fig. 1.** Examples of attack graphs and corresponding temporal attack graphs

*Example 1.* Consider the two attack graphs shown in Figures 1(a) and 1(b). In the one of Figure 1(a), both conditions $c_4$ and $c_5$ are required to execute exploit $v_4$. Condition $c_4$ can be achieved by either exploit $v_1$ or exploit $v_2$, whereas condition $c_5$ can only be achieved by exploit $v_3$. Thus $v_4$ must be preceded by $v_3$ and one of $v_1$, $v_2$. In the graph of Figure 1(b), only condition $c_4$ is necessary for exploit $v_4$, and it can be achieved by any of $v_1$, $v_2$, or $v_3$. The corresponding temporal attack graphs are shown in Figure 1(c) and 1(d) respectively.

An attack may be executed in multiple different ways, which we refer to as *instances* of the attack. Informally, an instance of a temporal attack graph $A$ is a tree $T = (V_T, E_T)$ over $A$, rooted at an end node of $A$. The root of the tree is an exploit implying the attack's target condition, whereas the leaves represent exploits depending on initial security conditions. Figure 2 shows the two possible instances of the temporal attack graph of Figure 1(c). Note that the execution of $v_3$ is always required to prepare for $v_4$, whereas only one of $v_1$, $v_2$ is required.

## 4 Attack Detection and Indexing

Intrusion alerts reported by IDS sensors typically contain several attributes. For the purpose of our analysis, we assume that each alert $o$ is a tuple ($type$, $ts$, $host_{src}$, $host_{dest}$), where $type$ denotes the event type, $ts \in \mathcal{T}$ is a timestamp, and $host_{src}$, $host_{dest}$ are the source and destination host respectively. We refer

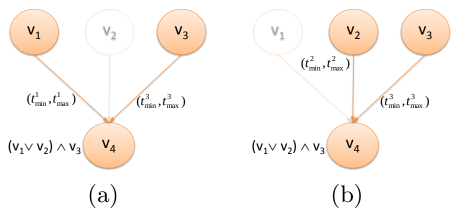


**Fig. 2.** Examples of temporal attack instances

to a sequence of such alerts as the *observation sequence*. Finally, we assume the existence of a function $\phi : L \to V$ that maps alerts to exploits (alerts that cannot be mapped to any known vulnerability are ignored). Given a temporal attack graph $A$ and an observation sequence $O$, we are interested in identifying sequences of alerts corresponding to instances of $A$. We refer to such sequences as *occurrences* of $A$.

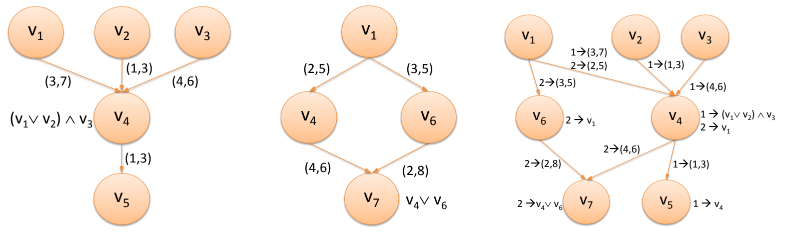
**Definition 2 (Temporal Attack Occurrence).** *Given an observation sequence* $O = \langle o_1, o_2, \ldots, o_n \rangle$ *and a temporal attack graph* $A = (V, E, \delta, \gamma)$*, an occurrence of* $A$ *in* $O$ *is a sequence* $O^* = \langle o^*_1, \ldots, o^*_k \rangle \subseteq O$ *such that:*

- $o^*_1.ts \le o^*_2.ts \le \ldots \le o^*_k.ts$;
- $\exists$ *an instance* $T = (V_T, E_T)$ *of* $A$ *such that* $V_T = \{v \in V \mid \exists o^* \in O^* \text{ s.t. } \phi(o^*) = v\}$*, i.e.,* $O^*$ *includes alerts corresponding to all the exploits in* $T$*;*
- $(\forall e = (v', v'') \in E_T)\ \exists o_{i'}, o_{i''} \in O^* \text{ s.t. } \phi(o_{i'}) = v' \wedge \phi(o_{i''}) = v'' \wedge t_{min} \le o_{i''}.ts - o_{i'}.ts \le t_{max}$*, where* $(t_{min}, t_{max}) = \delta(e)$.

*The* span *of* $O^*$ *is the time interval* $span(\langle o^*_1, \ldots, o^*_k \rangle) = [o_{i_1}.ts, o_{i_k}.ts]$. □

Note that multiple concurrent attacks generate interleaved alerts in the observation sequence. In order to concurrently monitor incoming alerts for occurrences of multiple types of attacks, we first merge all temporal attack graphs from $\mathcal{A} = \{A_1, \ldots, A_k\}$ into a single graph. We use $id(A)$ to denote a unique identifier for attack graph $A$ and $I_{\mathcal{A}}$ to denote the set $\{id(A_1), \ldots, id(A_k)\}$. Informally, a *Temporal Multi-Attack Graph* is a graph $G = (V_G, I_{\mathcal{A}}, \delta_G, \gamma_G)$, where $V_G$ is the set of all vulnerability exploits. A temporal multi-attack graph can be graphically represented by labeling nodes with vulnerabilities and edges with the id's of attack graphs containing them, along with the corresponding temporal windows. Note that the temporal multi-attack graph needs to be computed only once before building the index. Figure 3 shows two temporal attack graphs $A_1$ and $A_2$ and the corresponding temporal multi-attack graph.

**Definition 3 (Temporal Multi-Attack Graph Index).** *Let* $\mathcal{A} = \{A_1, \ldots, A_k\}$ *be a set of temporal attack graphs, where* $A_i = (V_i, E_i, \delta_i, \gamma_i)$*, and let* $G = (V_G, I_{\mathcal{A}}, \delta_G, \gamma_G)$ *be the temporal multi-attack graph built over* $\mathcal{A}$*. A* Temporal Multi-Attack Graph Index *is a 5-tuple* $I_G = (G, start_G, end_G, tables_G, completed_G)$*, where:*



(a) Attack graph $A_1$ (b) Attack graph $A_2$ (c) Multi-attack graph

**Fig. 3.** Example of temporal multi-attack graph

– *$start_G : V_G \rightarrow 2^{I_{\mathcal{A}}}$ is a function that associates with each node $v \in V_G$, the set of attack graph id's for which $v$ is a start node;*
– *$end_G : V_G \rightarrow 2^{I_{\mathcal{A}}}$ is a function that associates with each node $v \in V_G$, the set of attack graph id's for which $v$ is an end node;*
– *For each $v \in V_G$, $tables_G(v)$ is a set of records of the form ($current$, $attackID$, $ts_0$, $previous$, $next$), where $current$ is a reference to an observation tuple (alert), $attackID \in I_{\mathcal{A}}$ is an attack graph id, $ts_0 \in T$ is a timestamp, $previous$ and $next$ are sets of references to records in $tables_G$;*
– *$completed_G : I_{\mathcal{A}} \rightarrow 2^{\mathcal{P}}$, 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 attack occurrences.*$\square$

Note that $G$, $start_G$, $end_G$ can be computed a-priori, based on the set $\mathcal{A}$ of given attack graphs. All the index tables $tables_G$ are initially empty. As new alerts are received, index tables are updated accordingly, as described in Section 4.1. The index tracks information about which nodes are start and/or end nodes for the original attack graphs. For each node $v$, the index maintains a table $tables_G(v)$ that tracks partially completed attack occurrences, with each record pointing to an alert, as well as to previous and successor records. In addition, each record stores the time at which the partial occurrence began ($ts_0$).

## 4.1 Index Insertion Algorithm

This section describes an algorithm (Algorithm 1) to update the index when a new alert is received. The algorithm takes as input a temporal multi-attack graph index $I_G$, a new alert $o_{new}$ to be added to the index, and a boolean flag $f_{TF}$ indicating whether the Time Frame (TF) pruning strategy must be applied (we will explain the TF pruning strategy in Section 4.2).

Line 1 maps the newly received alert $o_{new}$ to a known vulnerability exploit $v_{new}$. Lines 3–7 handle the case when $v_{new}$ is the start node of an attack graph. A new record is added to $tables_G(v_{new})$ for every graph in $start_G(v_{new})$ (Lines 4-6), denoting the fact that $o_{new}$ may represent the start of a new attack sequence.

**Algorithm 1** $insert(o_{new}, I_G, f_{TF})$

```
Input: New alert to be processed $o_{new}$, temporal multi-attack graph index $I_G$, boolean flag $f_{TF}$
    indicating whether the Time Frame Pruning strategy must be applied.
Output: Updated temporal multi-attack graph index $I_G$.
1: $v_{new} \leftarrow \phi(o_{new})$ // Map the new alert to a known vulnerability exploit
2: // Look at start nodes
3: if $start_G(v_{new}) \neq \emptyset$ then
4:     for all $id \in start_G(v_{new})$ do
5:         add $(o_{new}^{\uparrow}, id, o_{new}.ts, \emptyset, \emptyset)$ to $tables_G(v_{new})$
6:     end for
7: end if
8: // Look at intermediate nodes
9: for all node $v \in V_G$ s.t. $\exists id \in I_{\mathcal{A}},\ \delta_G(v, v_{new}, id) \neq \texttt{null}$ do
10:     if $TF$ then
11:         $r_{first} \leftarrow \min\{r \in tables_G(v) \mid o_{new}.ts - r.current.ts \leq \max_{id \in I_{\mathcal{A}} \mid \delta_G(v,v_{new},id) \neq \texttt{null}} \delta_G(v, v_{new}, id).t_{max}\}$
12:     else
13:         $r_{first} \leftarrow tables_G(v).first$
14:     end if
15:     for all record $r \in tables_G(v)$ s.t. $r \geq r_{first}$ do
16:         $id \leftarrow r.attackID$
17:         if $\delta_G(v, v_{new}, id) \neq \emptyset$ then
18:             $(t_{min}, t_{max}) \leftarrow \delta_G(v, v_{new}, id)$
19:             if $(t_{min} \leq t_{new}.ts - r.current.ts \leq t_{max}) \wedge \gamma(v_{new})$ then
20:                 $r_n \leftarrow (o_{new}^{\uparrow}, id, r.ts_0, \{r^{\uparrow}\}, \emptyset)$
21:                 add $r_n$ to $tables_G(v_{new})$
22:                 $r.next \leftarrow r.next \cup \{r_n^{\uparrow}\}$
23:                 // Look at end nodes
24:                 if $id \in end_G(v_{new})$ then
25:                     add $r_n^{\uparrow}$ to $completed_G(id)$
26:                 end if
27:             end if
28:         end if
29:     end for
30: end for
```

Lines 9–30 look at the tables associated with the nodes that precede $v_{new}$ in the temporal multi-attack graph and check whether the new alert can be correlated to existing partially completed occurrences. For each predecessor $v$ of $v_{new}$, Lines 10-14 determine where the algorithm should start scanning $tables_G(v)$. Note that records are added to the index as new alerts are generated. Therefore, records $r$ in each index table $tables_G(v)$ are ordered by $r.current.ts$, i.e., the time at which the corresponding alert was generated. 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., $r_1.current.ts \leq r_2.current.ts$. In the unrestricted case, the whole table is scanned, $tables_G(v).first$ being the first record in $tables_G(v)$. If TF pruning is being applied, only the "most recent" records in $tables_G(v)$ are considered. On Lines 18–19, time intervals labeling the edges of temporal attack graphs are used to determine whether the new alert can be linked to record $r$ for the attack graph in $\mathcal{A}$ identified by $id = r.attackID$, given the amount of time elapsed between $r.current.ts$ and $o_{new}.ts$. Additionally, we verify whether the condition $\gamma(v_{new})$ labeling node $v_{new}$ is satisfied. If all these conditions are met, a new record $r_n$ is added to $tables_G(v_{new})$ and $r.next$ is updated to point to $r_n$ (Lines 20–22). Note that $r_n$ inherits $ts_0$ from its predecessor; 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 24–26 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 been completed (Line 25).

Algorithm *insert*, can be used iteratively for loading an entire observation sequence at once (we refer to this variant as *bulk-insert*).

### 4.2 Performance

In its unrestricted version, algorithm *bulk-insert*, has quadratic time complexity w.r.t. the size of the observation sequence, as typical of a *nested loop* procedure. However, we propose a pruning strategy – called *Time Frame* (TF) – that leverages the temporal constraints on the unfolding of attacks and lowers the complexity of bulk loading, while not altering the solutions. Algorithm *insert* is said to apply *Time Frame pruning* if, for each alert $o_{new}$ and each predecessor $v$ of $v_{new} = \phi(o_{new})$, it starts scanning $tables_G(v)$ at the first record $r$ such that $o_{new}.ts - r.current.ts \leq \max_{id \in I_{\mathcal{A}} \mid \delta_G(v, v_{new}, id) \neq \texttt{null}} \delta_G(v, v_{new}, id).t_{max}$.

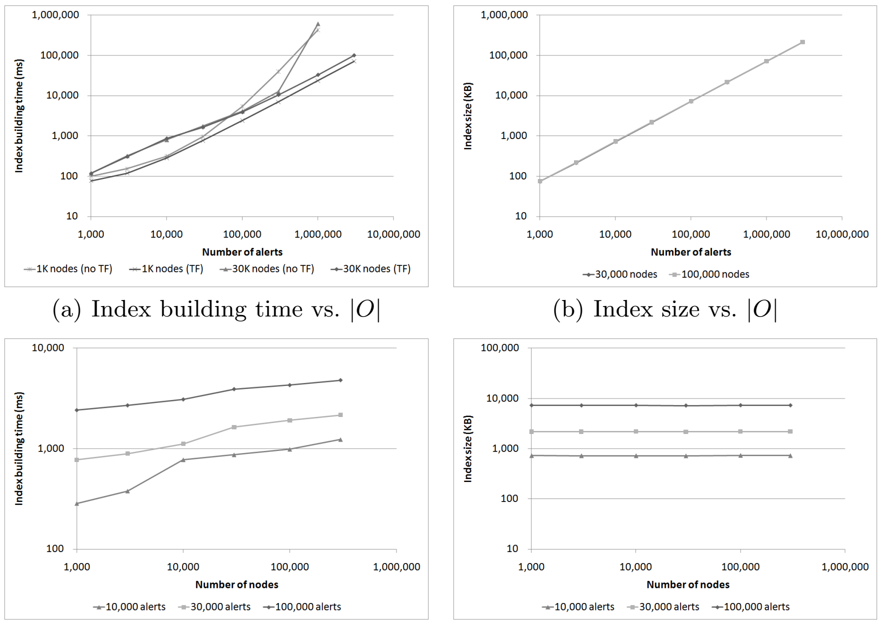
This strategy avoids scanning the entire predecessor table when most of the records in $tables_G(v)$ cannot be linked to records corresponding to $v_{new}$, because too long has passed since their corresponding alerts were generated. It can be proved that this strategy does not alter the search space, and the worst case complexity of the algorithm *insert* (resp. *bulk-insert*) when the TF pruning is applied is $O(k^{|V_G|} \cdot |\mathcal{A}|)$ (resp. $O(k^{|V_G|} \cdot |\mathcal{A}| \cdot |O|)$), where $O$ is the observation sequence and $k$ is the level of concurrency (i.e., maximum number of concurrent attacks). However, complexity is in practice lower than the worst case scenario and does not depend on the size of the graph, as experimentally shown in Section 5), since the number of index tables to be examined at each step is typically bounded and much smaller than $|V_G|$.

## 5 Preliminary Experiments

In this section, we report the results of the experiments we conducted to evaluate the time and memory performance of the proposed index.

We evaluated the system using synthetic datasets generated using two separate tools. The first tool generates random attack graphs by taking as input a set of vulnerabilities. The second tool generates alert sequences by simulating attacks described by the graphs generated using the first tool. We used the first tool to generate sets of attack graphs of varying size. We then used the second tool to generate sequences of 3 million alerts for each set of attack graphs.

Figure 4(a) shows how the time to build the entire index increases as the number of alerts increases. It is clear that, when TF is applied, the index building time is linear in the number of observations (note that both axes are on a log scale), and the bulk indexing algorithm can process between 25 and 30 thousands alerts per second. Also note that the size of the graphs does not significantly affects the index building time, as confirmed by Figure 4(c). In fact, when the size of the graphs changes by orders of magnitude, the processing time increases



(a) Index building time vs. $|O|$ (b) Index size vs. $|O|$

(c) Index building time vs. $|V_G|$ (d) Index size vs. $|V_G|$

**Fig. 4.** Experimental results

slightly, but remains in 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 $v \in V_G$) increases as well, but at the same time the average number of occurrences of each observation and the average size of each table decrease, keeping total memory occupancy (Figure 4(d)) and processing time (Figure 4(c)) almost constant. When TF is not applied, the index building time becomes quadratic, and starts to diverge. This effect is more evident for smaller graphs, as shown in Figure 4(a). This can also be explained by considering that, for a given number of alerts, when the size of the graphs decreases, there are fewer tables each containing a larger number of records, and the algorithm has to scan a constant number of such tables at each step. Finally, Figure 4(b) shows that memory occupancy is linear in the number of alerts, and confirms that it is independent from the size of the graphs.

In conclusions, we have shown that our framework can handle very large attack graphs, with hundreds of thousands of nodes, and can process incoming alerts at the rate of 25-30 thousands per second, which makes attack detection real-time for many real-world applications.

## 6 Conclusions and Future Work

In this paper, building on previous work on topological vulnerability analysis, we proposed an automated framework to manage very large attack graphs and monitor high volumes of incoming alerts for the occurrence of known attack patterns in real-time. Specifically, we proposed (i) a data structure to concurrently monitor multiple types of attacks; (ii) an index structure to effectively index millions of time-stamped alerts; and (iii) a real-time algorithm to process a continuous stream of alerts and update the index. Experimental results confirmed that our framework enables real-time attack detection.

## References

1. M. Albanese, A. Pugliese, V. S. Subrahmanian, and O. Udrea. MAGIC: A multi-activity graph index for activity detection. In *Proc. of the IEEE Intl. Conference on Information Reuse and Integration (IRI '07)*, pages 267–272, August 2007.
2. D. Avrahami-Zilberbrand, G. Kaminka, and H. Zarosim. Fast and Complete Symbolic Plan Recognition: Allowing for Duration, Interleaved Execution, and Lossy Observations. In *Proc. of the AAAI Workshop on Modeling Others from Observations (MOO-05)*, 2005.
3. O. Dain and R. K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proc. of the 2001 Workshop on Data Mining for Sec. App.*, pages 1–13, 2001.
4. T. Duong, H. Bui, D. Phung, and S. Venkatesh. Activity Recognition and Abnormality Detection with the Switching Hidden Semi-Markov Model. In *Proc. of IEEE CVPR-05*, volume 1, pages 838–845, 2005.
5. N. Habra, B. Charlier, A. Mounji, and I. Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In *ESORICS*.
6. R. Hamid, S. Maddi, A. Y. Johnson, A. F. Bobick, I. A. Essa, and C. L. I. Jr. A novel sequence representation for unsupervised analysis of human activities. *Artificial Intelligence*, 173(14):1221–1244, September 2009.
7. S. Jajodia and S. Noel. *Cyber Situational Awareness*, chapter Topological Vulnerability Analysis, pages 139–154. Springer, 2010.
8. S. Lühr, H. H. Bui, S. Venkatesh, and G. A. W. West. Recognition of human activity through hierarchical stochastic learning. In *Proc. of the 1st IEEE Intl. Conf. on Pervasive Computing and Comm. (PerCom-03)*, pages 416–422, 2003.
9. P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *Proc. of the 10th Conf. on Computer and Comm. Security (CCS '03)*, pages 200–209, 2003.
10. S. Noel, E. Robertson, and S. Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *Proc. of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, pages 350–359, 2004.
11. V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
12. X. Qin and W. Lee. Statistical causality analysis of INFOSEC alert data. In *Proc. of the 6th Intl. Symp. on Recent Advances in Intr. Detection (RAID '03)*, 2003.
13. A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proc. of the 4th Intl. Symp. on Recent Advances in Intr. Detection (RAID '01)*, pages 54–68, 2001.
14. L. Wang, A. Liu, and S. Jajodia. Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts. *Computer Comm.*, 29(15):2917–2933, 2006.