Antonino Rullo
Università della Calabria - DIMES

Cyber Defense of Enterprise Information Systems: Advanced Issues and Techniques

PhD Thesis

Advisors:
Andrea Pugliese
Domenico Saccà

Contents

1 **Introduction** ... 1

Part I Attack Detection

2 **Expressive and Efficient Online Alert Correlation** ... 7
2.1 Introduction ... 7
2.2 Preliminaries and Problem Formalization ... 9
2.2.1 Discussion ... 14
2.3 Efficient Retrieval ... 15
2.4 Experimental Results ... 19
2.4.1 Setting ... 19
2.4.2 Results ... 21
2.5 Related Works ... 25
2.5.1 Graph-Based Alert Correlation ... 25
2.5.2 Fusion-Based Alert Correlation ... 27
2.5.3 Pattern Discovery ... 27
2.5.4 Other Works ... 27
2.6 Conclusions ... 28

3 **Intrusion Detection with Hypergraph-Based Attack Models** 29
3.1 Introduction ... 29
3.2 Modeling Attack Processes ... 32
3.3 Consistency of Attack Models ... 35
3.4 The Intrusion Detection Problem ... 38
3.5 Action Hierarchies ... 40
3.6 Indexing and Detecting Attack Instances ... 41
3.7 Experimental Evaluation ... 45
3.7.1 Setting ... 47
3.7.2 Results ... 48
3.8 Related Work ... 51

3.8.1 Hypergraphs in security 51
3.8.2 Workflow modeling 52
3.8.3 Intrusion detection with attack graphs 52
3.9 Conclusions 53

Part II Anomaly Detection

4 PADUA: Parallel Architecture to Detect Unexplained Activities 57
4.1 Introduction 57
4.2 Related Work 59
4.2.1 A priori definitions 59
4.2.2 Learning and then detecting abnormality. 59
4.2.3 Similarity-based abnormality. 60
4.2.4 Cybersecurity. 60
4.3 Probabilistic Penalty Graphs 61
4.3.1 Definition of PPGs 61
4.3.2 Unexplained Situations 62
4.3.3 Deriving Noise Degradation Values from a Training Set 67
4.4 The PPG-Index: Fast Computation of Unexplained Situations on a Single CPU 68
4.4.1 Super-PPGs 68
4.4.2 The PPG-Index 69
4.4.3 Example: updating and pruning a PPG-Index 73
4.5 Partitioning Super-PPGs Across a Compute Cluster 74
4.5.1 Probability Partitioning (PP) and Probability-Penalty Partitioning (PPP) 75
4.5.2 Expected Penalty Partitioning (EPP) 76
4.5.3 Temporally Discounted Expected Penalty Partitioning (tEPP) 77
4.5.4 Occurrence Partitioning (OP) 78
4.6 Parallel Detection 78
4.7 Experimental Results 79
4.7.1 Video Surveillance Domain 80
4.7.2 Cybersecurity Domain 83
4.8 Conclusions 84

Part III Adversarial Defense

5 Pareto-Optimal Adversarial Defense of Enterprise Systems 89
5.1 Introduction 89
5.2 Related Work 91
5.3 Vulnerability Dependency Graphs 93

5.3.1 From Attack Graphs to Vulnerability Dependency Graphs . . . 95
5.4 Players' Strategy . . . 98
5.4.1 Defender's Strategy . . . 98
5.4.2 Attacker Strategy . . . 100
5.4.3 Best Strategy of the Attacker . . . 105
5.5 Pareto Analysis for the Defender . . . 105
5.5.1 Bi-Objective Optimization Problem Formulation . . . 106
5.5.2 Computing the Pareto Frontier . . . 110
5.5.3 Finding the Optimal Defender Strategy . . . 112
5.5.4 MILP Formulations for Bi-Optimization Problem and Optimal Defender Strategy . . . 113
5.5.5 Possible Extensions . . . 123
5.6 Experimental Results . . . 125
5.6.1 Topology of the VDGs Used in the Experiments . . . 125
5.6.2 Pareto Frontiers . . . 127
5.6.3 Execution Time . . . 130
5.7 Conclusion and Future Work . . . 131
5.8 Table of Symbols . . . 133

6 Conclusions . . . 135

References . . . 137

1

Introduction

The original purpose of the Internet was to be a data network dedicated to the spread of documents within the scientific community and not intended instead for purely public use. But the rapid growth of Internet occurred in the last 15 years has led many organizations, such as companies, governments, law enforcement, banks, universities, etc., to use the network as one of their most important tools. This made Internet perhaps the biggest existing interchange, trading and communication place, on which have migrated hundreds of services, and on which thousands of monetary transactions, trades and accumulation of information daily occur.

Today Internet is characterized as a democratic network where each single user can transmit information of any kind – often in a totally anonymous way. This fact, on the one hand, is one of the main reasons for the success of the network as a powerful tool for mass communication, but on the other, it generates inherent weaknesses and vulnerabilities. Indeed, many are the attacks achievable via the network from hackers with malicious intentions such as denial of service and theft of sensitive data (passwords and authentication codes, cloning of credit cards, etc.) as well as direct fraud against consumers such as through fake e-mail. The nature of Internet has therefore allowed the spread of a different kind of user, the malicious user, who uses the network infrastructure in order to cause damages to normal users or organizations.

The infrastructure of Internet presents a series of vulnerabilities, or flaws, which allow malicious users to bypass security controls, and to anonymously navigate a system along pathways not planned for normal users. Thus, organizations with network-connected information systems must necessarily put appropriate countermeasures in place in order to prevent malicious users from taking advantage of their systems. Industry has a very strong market-based incentive to ensure that networks are safe and secure, with participants working in partnerships with government and even with competitors. Protecting cyberspace is critical from the perspective of both industry and government. They share common interests in building confidence and security in the use

of information and communication technology (ICT) to support economic growth and national security.

Market pushes ICT firms to place a high priority on the security of their products and services. Moreover, as cyber attacks continue to increase in volume and sophistication, it is critical that the public and private sectors partner create a system with the flexibility to address threats as they evolve. Advanced Persistent Threat (APT) attacks have significantly changed the cyber threat landscape by introducing an adversary, likely backed by nation-states, with a high likelihood of success. With a high level of expertise, funding and organization, APT attackers will likely succeed in breaching a targeted system. In this scenario, there are three important security issues an organization needs to take care of:

- *Attack Detection*: making their systems able to recognize and deal with ongoing attacks when a malicious user has been able to begin one;
- *Anomaly Detection*: making their systems able to identify anomalous behaviors not necessarily classifiable as attacks;
- *Adversarial Defense*: making their systems immune to known attacks.

The objective of this thesis is to propose techniques that address the three above issues using advanced methods and techniques of two basic fields: model checking for intrusion and anomaly detection and game theory for adversarial defense.

In model checking, one develops a specification of an algorithm and then attempts to validate various assertions about the correctness of that specification under the specific assumptions about the model. Model checking provides a useful and rigorous framework for examining security issues.

Game theory has been used successfully in several areas. The approach explicitly models the interests of attackers and defenders and their repeated interactions. Game theory is useful in understanding how to prioritize cyber defense activities. It is not possible to protect everything all the time and so some notion of risk must be established; game theoretic approaches provide a framework for reasoning about such choices.

The thesis is divided into three parts that illustrate the problems dealt with for each of the mentioned issues, the adopted formal techniques and the main achieved results.

Part 1 on attack detection presents real-time intrusion detection techniques that, given a set of known attack patterns, index the "activities" that are happening in a monitored system in order to extract "attack instances", e.g., subsequences of the log that match some of the given patterns [1, 2, 3]. In particular, Chapter 1 describes a framework where known attacks are described by deterministic finite automatons equipped with correlation and severity functions that are used to further constrain the structure of attacks and assign them a severity value. During the detection process, attack instances returned by the pro- posed index structure and associated maintenance and detection algorithm are kept in a priority queue of size k, which orders them according to

their severity value so that only the k "more severe" (possibly ongoing) attacks are returned to the security expert. The framework presented in Chapter 2 uses hypergraph-based models to describe attacks. This provides the ability to describe attacks in a much more compact way, by allowing order-independent sets of action symbols that better capture the scenarios where some of the steps an attacker must follow do not necessarily have to be taken in a specific order.

Part 2 on anomaly detection proposes a novel technique that labels sub-sequences of the log as "unexplained" when they significantly differ from "explained" ones [4]. In particular, in Chapter 3 a graph-based model is proposed to describe activities that are "planned" to occur in the monitored system. Expected actions correspond to vertices, while edges describe probabilistically-weighted precedence relationships between actions. In this case, sub-sequences of the log that match a path in a model are given a "score" that is a function of the weights associated with the edges in the path. The model includes a "penalty" component, which is used to lower scores in the presence of noise in the log. A parallel detection algorithm merges graph models and extracts unexplained sub-sequences, based on a different definition of score aimed at capturing the degree of unexplainedness of the sub-sequences. The algorithm is designed to run over a $k + 1$ nodes cluster in which the merged graph is split according to some heuristics.

Finally, Part 3 on adversarial defense presents a defense technique that, given a set of software vulnerabilities, computes the Pareto-optimal sets of vulnerabilities that have to be patched in order to cover a portion of the network as wide as possible with limited resources [5]. The technique is developed with a game-theoretic approach. The game is a two-player game, and it is played once. Each player has a set of actions, and knows what the value of each action will be, but the value depends on what the other player does. So in principle each player considers all its options, based on what the other player might do, contemplating all its options. Each player adopts a strategy, possibly involving random choices, describing what they will do. There is more than one optimality concept. A Nash equilibrium occurs when neither player can do better just by changing their strategy. The two strategies together are Pareto optimal when every change that makes one player better off harms the other player. There can be multiple Nash equilibria, and multiple Pareto equilibria, and they need not agree. In particular, Chapter 4 proposes a technique for the defense of computer networks that, given a finite cost an organization is willing to bear to limit network security risks, and given a minimal "productivity level" the organization requires (associated with the possibility of deactivating some of the software deployed on the network) identifies the Pareto-optimal sets of patches to be applied. The idea is to model a scenario where we want to limit the chances for malicious users to cause damage, while keeping the total cost of the security operations below the available budget, and deactivating software in a way that the total associated productivity level does not decrease unacceptably.

Finally Chapter 6 draws the conclusion and discusses further research work.

Part I

Attack Detection

2

Expressive and Efficient Online Alert Correlation

We propose a technique for alert correlation that combines DFA-like patterns with correlation functions and supports the fast retrieval of occurrences of the given patterns through specifically-designed indexing and retrieval schemes. Our approach supports (i) the retrieval of the top-k (possibly non-contiguous) sub-sequences, ranked on the basis of an arbitrary user-provided severity function, (ii) the concurrent retrieval of sub-sequences that match any pattern in a given set, (iii) the retrieval of partial occurrences of the patterns, and (iv) the online processing of streaming logs. The experimental results confirm that, although the proposed model is very expressive, the indexing and retrieval schemes are able to guarantee a very high efficiency of the retrieval process.

2.1 Introduction

Intrusion Detection Systems (IDSs) usually generate logs whose tuples encode timestamped security-related alerts that are recorded from a monitored system. In general, the *alert correlation* process transforms groups of such alerts into *intrusion reports* of interest for the security expert. Alerts typically contain attributes like the type of event, the address of the source and destination hosts, etc. These attributes are matched against known vulnerabilities, in order to avoid reporting alerts wih no actual associated risk (e.g., a Linux-oriented attack blindly launched on a Windows machine). However, applying this approach alone can lead to missing relevant alerts that do not match any vulnerability (e.g., ICMP PINGs) but that can be part of a more complex multi-step attack. Alerts must therefore also be correlated using the knowledge encoded in specific structures (e.g. *attack graphs* [6]) that describe logical connections of interest among correlated alerts. In *anomaly detection* systems [4, 7, 8, 9, 10, 11], historical data is used to build profiles of the "normal" user behaviors, so that sequences of actions that deviate from the profiles are classified as "anomalous". *Misuse detection* systems [6, 12, 13, 14, 15, 16, 17] make

instead use of sets of descriptions of suspicious activities that are matched against the log in order to identify ongoing activities.

In order to describe logical connections among alerts, *multi-step* and *fusion-based* correlation techniques have been used in the past [18]. Multi-step correlation [19, 20, 21] seeks to identify suspicious activities that consist of multiple "steps" by modeling activities through attack graphs [6, 15, 22, 23, 24, 25] or deterministic finite automata (DFAs). Any activity that complies with a graph or a DFA description is considered suspicious. Fusion-based correlation [19, 20, 26] uses instead similarity functions that, when applied to the attributes of incoming alerts, establish whether they should be considered part of a same activity.

The current literature about alert correlation provides a wide variety of methods based on the above approaches. In this chapter, we propose a technique that combines DFA-like patterns and correlation functions, and supports the fast retrieval of occurrences of the given patterns using specifically-designed indexing and retrieval schemes. Our approach provides the following main features:

- The objective is that of retrieving the *top-k sub-sequences* of a log (ranked on the basis of a user-provided *severity function*) that match some graph-based pattern and satisfy the constraints expressed by a user-provided *correlation function*.
- We *do not mandate any specific schema for the alerts*: we simply regard each alert as a relational tuple with a user-provided schema.
- Both the user-provided correlation and severity functions can be *arbitrary* – we only mandate their polynomial-time computability.
- The proposed indexing and retrieval schemes are designed to manage *multiple patterns*, each with its specific severity and correlation functions.
- The retrieved sub-sequences can possibly be *non-contiguous*.
- The user can specify a *maximum duration* for each pattern, in order to retrieve only the sub-sequences that fit in specific time windows.
- The reports built can be based on *partial occurrences* of activities of interest, i.e., sub-sequences that have not yet reached their terminal stages in the DFAs.
- The log is *streamed* into the system, so the retrieval of correlated alerts is performed in an *online* fashion with appropriate tuple rates (i.e., tuples processed per second).

Figure 2.1 shows the two patterns we will use as our running example throughout the chapter. Edges are labeled with alert symbols and each stage is annotated with the severity value associated with it. The sequence {*access*, *service exploit*, *DoS*} represents a possible *Denial of Service* attack. A security expert may want to take security measures at a certain "depth" of this attack. To this end, the expert wants to receive a report every time a stage of the sequence is traversed. In other words, we must look at all sub-sequences of the log that match some prefix of any path in the pattern. Furthermore,

in order to counter the intrusions more quickly, the expert may want to only look at the first k sub-sequences, based on their associated severity value – in the example, we assume that the severity of a sub-sequence only depends on the stage reached in the pattern. Moreover, the correlation function looks at the attributes of the alerts in order to decide which alerts are to be considered part of a same attack. Finally, for each pattern, only the sequences that fit in a time window of maximum length τ are considered.

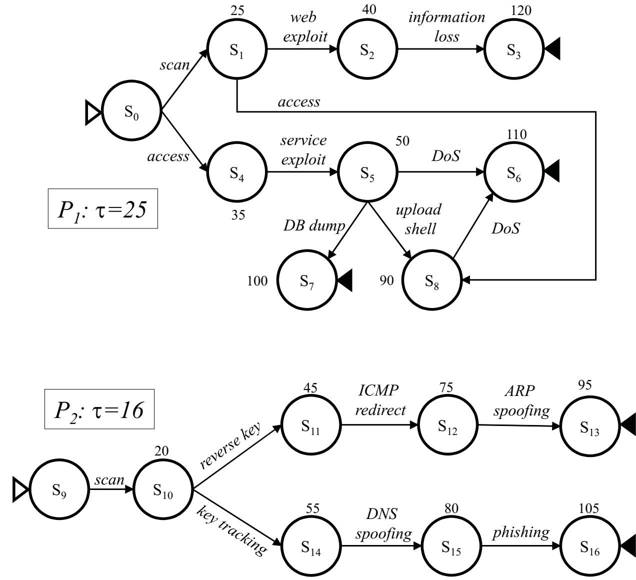


Fig. 2.1. Example patterns. Each stage is annotated with its associated severity value.

2.2 Preliminaries and Problem Formalization

In this section we introduce some preliminary notions and formalize the alert correlation problem, which basically consists in finding the top-k sub-sequences of a log that represent an attack w.r.t. a given set of patterns.

We assume the existence of (i) a finite set $\mathcal{A}$ of *alert* symbols and (ii) w *attribute domains* $ATT_1, \ldots, ATT_w$. A *log* is a set $L = \{\ell_1, \ldots, \ell_n\}$ of tuples (each corrisponding to an alert) of the form $\langle id, symbol, ts, att_1, \ldots, att_w \rangle$ where id is an identifier, $symbol \in \mathcal{A}$, $ts \in \mathbb{N}$ is a timestamp, and $\forall i \in [1, w]$, $att_i \in ATT_i$. We assume that $\forall i \in [1, n-1]$, $\ell_i.ts < \ell_{i+1}.ts$. Moreover, we denote component c of log tuple ℓ as $\ell.c$.

The notion of a pattern is formalized by the following definition.

Definition 2.1 (Pattern). *A* pattern *is a tuple* $P = \langle S, s_s, S_t, \delta, \tau \rangle$ *where:*

- S *is a set of stages;*
- $s_s \in S$ *is the start stage;*
- $S_t \subseteq S$ *is the set of terminal stages;*
- $\delta : S \times \mathcal{A} \rightarrow S$ *is the stage transition (partial) function;*[1]
- $\tau \in \mathbb{N}$ *is the maximum duration of an occurrence of* P.

We assume that $\forall s \in S_t, \forall sym \in \mathcal{A}$, $\delta(s, sym)$ *is not defined, and that* $\forall s \in S, \forall sym \in \mathcal{A}$, $\delta(s, sym) \neq s_s$.

In the following, when $\delta(s, sym) = s'$, we say that there is an edge from s to s' labeled with sym.

Example 2.2. Pattern P_1 of our running example is formalized as follows:

- $S = \{s_0, \ldots, s_8\}$;
- $s_s = s_0$;
- $S_t = \{s_3, s_6, s_7\}$;
- $\delta(s_0, scan)=s_1$, $\delta(s_0, access)=\delta(s_1, access)=s_8$, $\delta(s_1, web\ exploit)=s_2$, $\delta(s_2, information\ loss)=s_3$, $\delta(s_4, service\ exploit)=s_5$, $\delta(s_5, DoS)=\delta(s_8, DoS)= s_6$, $\delta(s_5, DB\ dump)=s_7$, $\delta(s_5, upload\ shell)=s_8$;
- $\tau = 25$.

An occurrence of a given pattern is a possibly non-contiguous subsequence of the log whose associated alert symbols correspond to a path that begins in a start stage. In addition, the overall duration of the subsequence must comply with the maximum duration allowed by the pattern. The following definition formalizes this.

Definition 2.3 (Occurrence). *Given a pattern* $P = \langle S, s_s, S_t, \delta, \tau \rangle$ *and a log* L*, an* occurrence *of* P *in* L *is a set* $O = \{\ell_1, \ldots, \ell_m\} \subseteq L$ *such that:*

- $\forall i \in [1, m-1]$, $\ell_i.ts < \ell_{i+1}.ts$;
- *there exists a set* $\{s_0, s_1, \ldots, s_m\} \subseteq S$ *such that:*
  - $s_s = s_0$;
  - $\forall i \in [1, m]$, $\delta(s_{i-1}, \ell_i.symbol) = s_i$;
- $\ell_m.ts - \ell_1.ts \leq \tau$.

[1] Some past works assume aciclicity of the patterns because, in many practical cases, the “criticality” associated with a sequence of alerts does not change when the sequence contains a portion that is repeated multiple times as it matches a cycle in the pattern. In such cases, the overall sequence is equivalent to the one obtained after removing the portion matching the cycle. We do not make this assumption as it would reduce the expressiveness of the model and it is not needed by the indexing algorithm we discuss in Section 2.3.

It should be observed that Definition 2.3 does not require an occurrence to reach a terminal stage. This feature gives security experts complete freedom in deciding whether or not a certain subsequence must be considered "critical" (i.e., with a high severity). Thus, any prefix of a complete path in the pattern can correspond to a critical subsequence the framework must take into account. Terminal stages are used to semantically represent the "final goal" of the attacker. Moreover, they help the retrieval algorithm as they signal that a subsequence can no longer be extended.

The following definition formalizes the way we characterize the severity of a subsequence and the attribute-based correlation among log tuples.

Definition 2.4 (Severity and Correlation Functions). *Given a pattern P and a log L, the* severity w.r.t. P *is a function*

$$\sigma_P : 2^L \rightarrow \mathbb{N}.$$

Moreover, the correlation w.r.t. P *is a function*

$$\gamma_P : 2^L \rightarrow \{true, false\}$$

such that $\gamma_P(X) = true$ for all subsets $X \subseteq L$ that, based on their attribute values, can be part of a same occurrence.

We assume transitivity of function γ_P, that is, if $\gamma_P(X_1 \cup X_2) = true$ and $\gamma_P(X_2 \cup X_3) = true$, then $\gamma_P(X_1 \cup X_3) = true$. Thus, we will sometimes use the equivalent notation $\gamma_P : L \times L \rightarrow \{true, false\}$. It should also be observed that it is natural to assume $\sigma_P(X) = 0$ when X is not an occurrence of P in L.

We are now ready to define the alert correlation problem we address.

Definition 2.5 (Alert Correlation Problem). *Given a set $\mathcal{P}$ of patterns, a log L, and a number $k \in \mathbb{N}$, the* alert correlation problem *consists in finding a set $\mathcal{O} = \{O_1, \ldots, O_k\}$ such that:*

1. *each O_i is an occurrence in L of a pattern $P_i \in \mathcal{P}$;*
2. *$\forall i \in [1, k]$, $\gamma_{P_i}(O_i) = true$;*
3. *$\forall i \in [1, k-1]$, $\sigma_{P_i}(O_i) \geq \sigma_{P_{i+1}}(O_{i+1})$;*
4. *there do not exist a pattern $P \in \mathcal{P}$ and an occurrence $O \notin \mathcal{O}$ of P in L such that $\sigma_P(O) > \sigma_{P_k}(O_k)$.*

In Definition 2.5, Condition 2 states that all tuples in each occurrence $O_i \in \mathcal{O}$ must be correlated to one another; Condition 3 states that $\mathcal{O}$ contains occurrences in decreasing order of severity value; Condition 4 ensures that the occurrences in $\mathcal{O}$ are the ones with the top-k severity values. We do not assume that $\forall i, j$ with $i \neq j$, $P_i \neq P_j$ – in other words, set $\mathcal{O}$ can contain two different occurrences of the same pattern.

It should be noted that if the security expert is only interested in contiguous occurrences (as the majority of existing approaches do), our proposed framework can be straighforwardly extended to post-process the retrieved occurreces and filter out non-contiguous ones.

Example 2.6. Returning to our running example, suppose we want to find the occurrences of the patterns in the log of Figure 2.2. In this case, log tuples are of the form $\langle id, symbol, ts, sourceIP, targetIP\rangle$. We assume that γ_{P_1} and γ_{P_2} consider log tuples as correlated if their *sourceIP*s are equal and their *targetIP*s are in the same subnetwork w.r.t. the example network in Figure 2.3. Moreover, σ_{P_1} and σ_{P_2} return the values in Figure 2.1 if the *targetIP*s of the tuples are outside the firewall – values are doubled if the *targetIP*s are inside the firewall. The resulting sub-sequences are listed in Figure 2.4, ordered by severity value. Note that O_8 is not an occurrence of P_2

id	symbol	ts	sourceIP	targetIP
100	scan	12	160.57.91.110	110.80.70.120
101	reverse key	13	160.57.91.110	110.80.70.120
102	scan	14	130.10.71.151	120.15.62.140
103	buffer overflow	15	190.23.41.170	170.21.88.124
104	web exploit	16	130.10.71.151	120.15.62.141
105	SQL injection	24	190.23.41.170	170.21.88.124
106	information loss	26	190.23.41.170	170.21.88.124
107	ICMP redirect	28	160.57.91.110	110.80.70.122
108	ARP spoofing	29	160.57.91.110	110.80.70.129
109	DoS	32	190.23.41.170	170.21.88.124
110	information loss	35	130.10.71.151	120.15.62.146

Fig. 2.2. Example log.

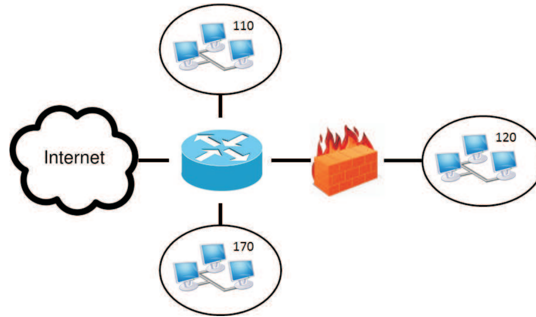


Fig. 2.3. Example network.

according to Definition 2.3, since its duration is 17 time units which is longer than the maximum duration of any occurrence of P_2 (that is, 16 time units).

Sub-sequence	Pattern	Severity	Duration
$O_1 = \{102, 104, 110\}$	P_1	240	21
$O_2 = \{102, 104\}$	P_1	80	2
$O_3 = \{100, 101, 107\}$	P_2	75	16
$O_4 = \{100, 101\}$	P_2	45	1
$O_5 = \{102\}$	P_1	50	0
$O_6 = \{100\}$	P_2	25	0
$O_7 = \{100\}$	P_1	20	0
$O_8 = \{100, 101, 107, 108\}$	P_2	0	17

Fig. 2.4. Example sub-sequences (only log tuple ids are shown).

The set $\mathcal{O} = \{O_1, \ldots, O_4\}$ is a solution for the alert correlation problem with $k = 4$[2]. In fact, it satisfies all of the conditions of Definition 2.5:

1. each O_i has an associated pattern P_i for which it is an occurrence in L;
2. $\gamma_{P_1}(O_1) = \gamma_{P_2}(O_2) = \gamma_{P_3}(O_3) = \gamma_{P_4}(O_4) = true$;
3. $\sigma_{P_1}(O_1) \geq \sigma_{P_1}(O_2) \geq \sigma_{P_2}(O_3) \geq \sigma_{P_2}(O_4)$;
4. $\sigma_{P_1}(O_5) \leq \sigma_{P_2}(O_4)$ and $\sigma_{P_2}(O_6) \leq \sigma_{P_2}(O_4)$.

In the characterization of the complexity of the alert correlation problem we target, we make the realistic assumption that the computation of functions γ and σ can be performed in polynomial time. We therefore denote the complexity of computing such functions as $O(poly_{\gamma,\sigma}(x))$, that is a polynomial in the cardinality x of the set to which the functions are applied. The following result establishes the overall complexity of the problem.

Proposition 2.7. *The worst-case asymptotical time complexity of solving the alert correlation problem is*

$$\Omega\left(log\ k \cdot \sum_{P=\langle S,s_s,S_t,\delta,\tau\rangle\in\mathcal{P}} (\tau^{|S|} \cdot poly_{\gamma,\sigma}(\tau))\right).$$

To see why the above result is true, it suffices to observe that:

1. τ is the maximum cardinality of an occurrence of P, so $\tau^{|S|}$ is the maximum possible number of occurrences of P in L. It should be observed that the existence of a "local time window" where alerts can be "connected" is common to all the models that allow to constrain the length of the sub-sequences (see, e.g., [16]) – obviously, without such constraints, this term would become $|L|^{|S|}$.
2. $poly_{\gamma,\sigma}(\tau)$ represents the time needed to check the correlation among the tuples of an occurrence of P and to compute their severity.

[2] Note that a security expert may want to discard O_2 and O_4 because they are prefixes of O_1 and O_3 respectively.

3. To extract the top-k occurrences, it suffices to maintain a priority queue of maximum size k while scanning the whole set of occurrences – this takes time $log\ k$ for each occurrence.

2.2.1 Discussion

To complete this section, we provide examples of how two interesting models proposed in the past can be expressed using our model.

In [22] the authors formally define an attack graph 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 $R_i \circ R_r$.

The problem of looking for all the occurrences of an attack graph $G = (V \cup C, R_r \cup R_i)$ in a log that reports security exploits is fully equivalent to our alert correlation problem if we translate G into a pattern $P = \langle S, s_s, S_t, \delta, \tau \rangle$ along with functions γ_P and σ_P as follows (where we denote the label of an exploit $v \in V$ as $label(v)$).

- We define $S = \{s_0\} \cup \{s_{label(v)} \mid v \in V\}$, $s_s = s_0$, and $S_t = \{s_{label(v)} \mid v$ is an end node in $G\}$.
- We define function δ so that if $(v, v') \in R_i \circ R_r$ then $\delta(s_{label(v)}, label(v')) = s_{label(v')}$. Moreover, if $v \in V$ is a start node in G, then $\delta(s_0, label(v')) = s_{label(v')}$.
- We set $\tau = k = \infty$.
- We define $\gamma_P(O)$ so that it evaluates to *true* iff O corresponds to a path in G from a start node to an end node.
- We define $\sigma_P(O)$ so that it evaluates to a fixed value $\sigma' > 0$ if O corresponds to a path in G from a start node to an end node, and zero otherwise.

A more complex stochastic temporal automaton-based model is proposed in [16]. Here, the edges in an automaton A are annotated with probabilities, which in turn depend on the time elapsed between two consecutive *observations* (each log tuple corresponds to an observation). More specifically, each edge is annotated with a *timespan distribution* that is a pair (I, τ) where I is a set of time intervals and $\tau : I \rightarrow [0, 1]$ is a function that associates a value $\tau(x, y) \in [0, 1]$ with each time interval $[x, y] \in I$. Then, an occurrence O of the automaton is defined as a sequence of log tuples matching a path from a start to an end node in A, and its overall probability $prob(O)$ is the product of the probabilities between all the pairs of consecutive observations in O. In addition, O is considered valid w.r.t. the "context" of its tuples (expressed through a *context* attribute) iff for any $t, t' \in O$, $t.context \simeq t'.context$, where $\simeq$ is an equivalence relation defined over the domain of the *context* attribute.

The *evidence problem* defined by the authors consists in finding all occurrences in a log having a probability higher or equal to a given threshold p. Given an automaton A, we can build a pattern P that makes the evidence problem solvable based on the solution to our alert correlation problem. In this case, we define the components of P and the value of k as in the previous case. Then we define:

- $\gamma_P(O)$ so that it evaluates to $true$ iff $\forall t, t' \in O$, $t.context \simeq t'.context$.
- $\sigma_P(O) = prob(O)$ if O corresponds to a path in G from a start node to an end node, and zero otherwise.

It is easy to see that, after computing the solution to our alert correlation problem, it suffices to select all the occurrences O having $\sigma_P(O) \geq p$ to obtain a correct solution to the evidence problem.

2.3 Efficient Retrieval

In this section we introduce a data structure called AC-Index, whose objective is that of efficiently "tracking" the occurrences of a given set of patterns in a log. The index is updated as soon as a new log tuple enters the system, and it contains a priority queue whose content represents the top-k occurrences found so far in the log.

We denote the set of patterns as $\mathcal{P}$. Without loss of generality, we assume $\bigcap_{\langle S,s_s,S_t,\delta,\tau\rangle \in \mathcal{P}} S = \emptyset$. Moreover, we use $\mathcal{S}$ to denote the set $\bigcup_{\langle S,s_s,S_t,\delta,\tau\rangle \in \mathcal{P}} S$. Finally, given an alert symbol $sym \in \mathcal{A}$, we define $stages(sym) \subseteq \mathcal{S}$ as the set of non-terminal stages having an incoming edge labeled with sym — formally, $\forall s \in stages(sym)$, $\exists \langle S, s_s, S_t, \delta, \tau\rangle \in \mathcal{P}$ such that $s \in S$, $s \neq s_s$, $s \notin S_t$, and $\exists s' \in S$ such that $\delta(s', sym) = s$.

We are now ready to define our AC-Index.

Definition 2.8 (AC-Index). *Given a set $\mathcal{P}$ of patterns an a log L, an AC-Index $I_{\mathcal{P}}$ is a tuple $\langle Tables, MainTable, PQ\rangle$ where:*

- *$Tables$ is a set containing a table $table(s)$ for each $s \in stages(sym)$ with $sym \in \mathcal{A}$. $table(s)$ contains rows of the form (PL, sev) where PL is a list of pointers to tuples in L, and $sev \in \mathbb{N}$ is the severity value corresponding to the set of tuples pointed by PL;*
- *$MainTable$ is a table where each row is of the form (sym, Z), where $sym \in \mathcal{A}$ and Z is a set of pointers to tables $table(s)$;*
- *PQ is a priority queue containing pairs of the form (PL, sev) that are copies of table rows in $tables(s)$. The size of PQ is bounded by k and the priority is the value of sev.*

In the AC-Index, a row $(PL = \{\ell_0^{\uparrow}, \ldots, \ell_m^{\uparrow}\}, sev) \in table(s)$ corresponds to an occurrence $O = \{\ell_0, \ldots, \ell_m\}$ in L of a pattern $P = \langle S, s_s, S_t, \delta, \tau\rangle \in \mathcal{P}$ with $sev = \sigma_P(O)$ and $\delta(s', \ell_m.symbol) = s$ for some $s' \in S$. Following the

definition of set *stages*, no table is built for neither initial stages (because such stages cannot correspond to occurrences) nor terminal stages (because we do not need to store non-extendable occurrences). In $MainTable$, a row (sym, Z) encodes the fact that, for each table $tables(s)$ pointed by Z there exists a stage $s \in \mathcal{S}$ with at least one ingoing edge labeled with sym. Finally, PQ always contains the k occurrences found so far with higher severity values. Moreover, if requested by the security expert, PQ can be configured in such a way that it will discard the occurrences that are prefixes of some occurrence of the same pattern. In our running example, O_2 and O_4 would be discarded since they are prefixes of O_1 and O_3, respectively.

Example 2.9. Figure 2.5 shows the initial status of the AC-Index built over pattern $P_1 = \langle S, s_s, S_t, \delta, \tau \rangle$. At this stage, PQ and all $table(s)$ are empty. $MainTable$ contains a number of rows equal to the number of distinct alert symbols labeling edges that end in non-terminal stages.

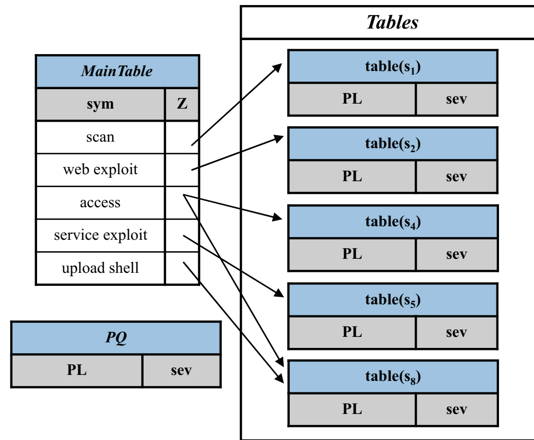


Fig. 2.5. Example initial index status.

Figure 3.7 shows the pseudo-code of the Insert algorithm that indexes a new log tuple ℓ_{new} with associated alert symbol $\ell_{new}.symbol$.

In the algorithm, Lines 6-9 deal with the case where s is a start stage, by creating a new occurrence. Specifically, it creates a new row table r and adds it to PQ and to $table(s')$, where s' is the stage reached from s by following the edge labeled with sym. Lines 11-20 check whether the new log tuple ℓ_{new} can be correlated with those in the existing occurrences. If it does (Lines 13-20), it is appended to such occurrences and the latter are added to PQ. Otherwise, i.e., if it does not fit in the time window τ, then the last log tuple of each occurrence that can not be extended is removed from its related table (Line 22). Observe that this implicitly corresponds to a pruning process that

```
    Algorithm: Insert(ℓ_new, I_P)
    Input: New log tuple ℓ_new, AC-Index I_P = ⟨Tables, MainTable, PQ⟩
    Output: Updated AC-Index I_P
1   sym ← ℓ_new.symbol
2   Z ← MainTable(sym)
3   for each table(s) ∈ Z
4     let P = ⟨S, s_s, S_t, δ, τ⟩ be a pattern s.t. s ∈ S
5     s' ← δ(s, sym)
6     if s = s_s
7       r ← (ℓ↑_new, 0)
8       if s' ∉ S_t then add r to table(s')
9       add r to PQ
10    else
11      for each row r ∈ table(s)
12        let O = {ℓ_1, ..., ℓ_n} be the set of log tuples pointed by r.PL
13        if ℓ_new.ts − ℓ_1.ts ≤ τ
14          if γ_P({ℓ_n, ℓ_new})
15            O' ← append ℓ_new to O
16            PL' ← append ℓ↑_new to r.PL
17            r' ← (PL', σ_P(O'))
18            if s' ∉ S_t then add r' to table(s')
19            add r' to PQ
20          end if
21        else
22          remove r from table(s)
23        end if
24      end for
25    end if
26  end for
```

Fig. 2.6. Insert algorithm.

is applied during the construction of the index, as opposed to the concurrent pruning applied by the Prune algorithm, which will be discussed later on.

Example 2.10. Figure 2.7 shows the status of the AC-Index after indexing log tuples from 102 to 110 of our running example when considering pattern $P_1 = \langle S, s_s, S_t, \delta, \tau\rangle$ only.

The indexing process can be divided into 3 distinct macro-steps:

1. The first processed log tuple is $\langle \mathit{102}, scan, \ldots\rangle$. Since there exists a row $(scan, \{table(s_1)^{\uparrow}\})$ in $MainTable$, row $r_1 = (PL = [\mathit{102}^{\uparrow}], sev = 50)$ is added to $table(s_1)$ (50 is the severity value returned by σ_{P_1}). Then, a copy of r_1 is added to PQ. Log tuple $\langle$*103, buffer overflow,...*$\rangle$ is skipped becaue there are no rows in $MainTable$ with $sym =$ *buffer overflow*.
2. Log tuple $\langle \mathit{104}, \ \mathit{web\ exploit}, \ldots\rangle$ can be correlated with $\langle \mathit{102}, scan, ...\rangle$, because $\delta_{P_1}(s_1, scan) = s_2$ and $\gamma_{P_1}(102, 104) = true$. Thus, row $r_2 = ([\mathit{102}^{\uparrow}, \mathit{104}^{\uparrow}], 80)$ is added to $table(s_2)$ and PQ. Log tuples from 105 to 109

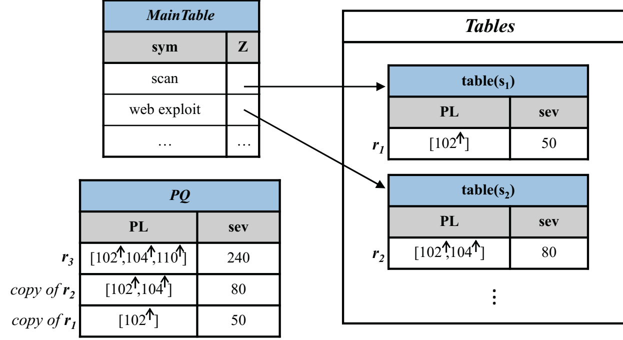


Fig. 2.7. Example index status after indexing log tuples from 102 to 110 of the log of Figure 2.2.

are skipped because none of them can be correlated with log tuples 102 or 104. As an example, tuple $\langle 106, \mathit{information\ loss}, \ldots\rangle$ cannot be linked to the occurrence O={*102, 104*} although $\delta_{P_1}(s_2, \mathit{information\ loss}) = s_3$, because $\gamma_{P_1}(\{102, 104, 106\}) = \mathit{false}$ due the $\mathit{IPAttacker}$ attribute value, which is distinct from that of tuples 102 and 104.

3. Log tuple $\langle 110, \mathit{information\ loss}, \ldots\rangle$ can be correlated with $\{102, 104\}$, because $\delta_{P_1}(s_2, \mathit{information\ loss}) = s_3$ and $\gamma_{P_1}(104, 110) = \mathit{true}$. However, in this case, a new row $r_3 = ([\mathit{102}^{\uparrow}, \mathit{104}^{\uparrow}, \mathit{110}^{\uparrow}], 240)$ is directly added to PQ because there does not exist $\mathit{table}(s_3)$ since s_3 is a terminal stage.

As the example shows, we only need to store occurrences in Tables if they can be extended. In fact, when an occurrence ends in a terminal stage it is no longer extendable, so it can be directly stored in PQ – this is why the AC-Index does not contain any $\mathit{table}(s)$ with s being a terminal stage.

The following result ensures that Algorithm Insert solves the alert correlation problem both correctly and optimally.

Proposition 2.11. *Given a log L, the execution of Algorithm Insert on all tuples in L terminates, and after the execution, the content of PQ represents the correct solution to the alert correlation problem. The worst-case asymptotical time complexity of Insert is*

$$O\left(\mathit{log}\ k \cdot \sum_{P=\langle S, s_s, S_t, \delta, \tau\rangle \in \mathcal{P}} (\tau^{|S|} \cdot \mathit{poly}_{\gamma,\sigma}(\tau))\right).$$

The Insert algorithm can work in parallel with the Prune algorithm shown in Figure 2.8, that updates Tables by linearly scanning them and removing every row which represents an occurrence that is no longer extendable based on the maximum duration τ. This pruning process saves both memory space and processing time – moreover, it can be performed in parallel with the Insert algorithm.

```
  Algorithm: Prune(I_P, ts)
  Input: AC-Index I_P = ⟨Tables, MainTable, PQ⟩, current timestamp ts
  Output: Updated AC-Index I_P
1 for each table(s) ∈ Tables
2   let P = ⟨S, s_s, S_t, δ, τ⟩ be the pattern s.t. s ∈ S
3   for each r ∈ table(s)
4     let {ℓ_1, ..., ℓ_n} be the set of log tuples pointed by r.PL
5     if ts − ℓ_1.ts > τ
6       remove r from table(s)
7     end if
8   end for
9 end for
```

Fig. 2.8. Pruning algorithm.

It should be noted that the pruning policy could also vary based on the structure of the severity function σ. In fact, a different pruning algorithm could evaluate in advance the maximum severity an occurrence can reach in the future, and immediately prune it if its maximum severity is under the current k-th. This policy could for instance be applied to our running example, where the severity only depends on the stage reached by the occurrence.

The following result ensures the correctness of the Prune algorithm.

Proposition 2.12. *Given a log $L = \{\ell_1, \ldots, \ell_n\}$, let $I^0_{\mathcal{P}}$ be the empty AC-Index and $I^i_{\mathcal{P}}$ be the AC-Index returned by Insert$(\ell_i, I^{i-1}_{\mathcal{P}})$. For any $i \in [1..n]$, the set of solutions in $I^i_{\mathcal{P}}$ is equal to the one in Prune$(I^i_{\mathcal{P}}, \ell_i.ts)$.*

2.4 Experimental Results

In this section we report on the experimental assessment we performed on our proposed alert correlation framework when applied to real-world and synthetic patterns. We implemented the framework in Java and run the experiments on an Intel Core i7-3770K CPU clocked at 3.50GHz, with 12GB RAM, running Windows 8.

2.4.1 Setting

We ran two rounds of experiments:

1. In the first round, we used real-world patterns P_1 and P_2 of Figure 2.1 and P_3 and P_4 of Figure 2.9.
2. In the second round, we used synthetic patterns P_5 and P_6 of Figure 2.10.

For both rounds, we built synthetic logs consisting of 300K tuples. Each log was built by combining a set of sub-logs, each of which is a sequence of alert symbols that can represent an occurrence of a given pattern. Specifically, a log

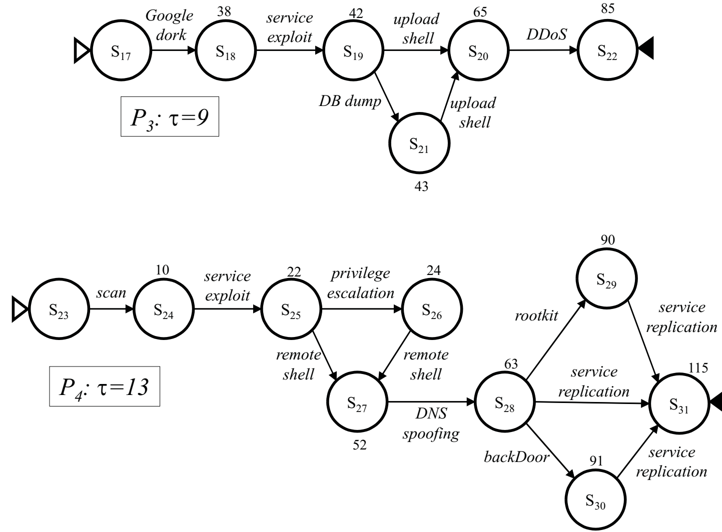


Fig. 2.9. Real-world patterns P_3 and P_4.

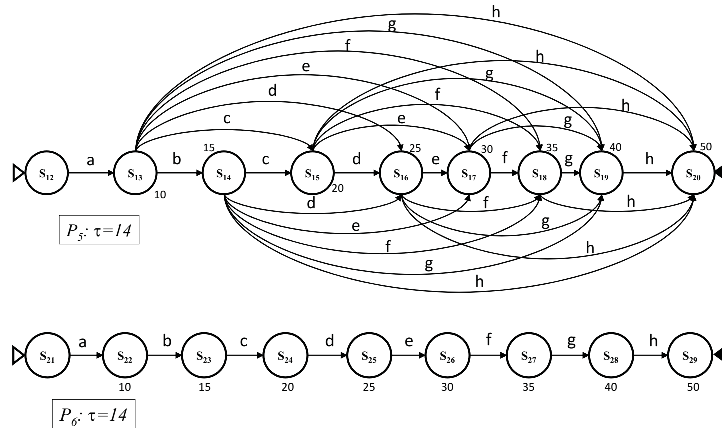


Fig. 2.10. Synthetic patterns P_5 and P_6.

combines several sub-logs $\{L_1, \ldots, L_n\}$ where each L_i is built by considering a path from an initial to a terminal stage in a pattern. These sub-logs were built and combined under six different *log generation modes*, each corresponding to a possible real-world scenario:

1. each sub-log only contains alert symbols in its corresponding pattern, and the sub-logs are concatenated;
2. same as mode 1, except that some alert symbols are *substituted* with "noise", i.e. with symbols not present in the corrisponding pattern, with a certain frequency;

3. same as mode 1, except that noise is *inserted* in the sequence;
4. same as mode 1, except that a certain percentage of each L_i partially overlaps with L_{i+1};
5. same as mode 2, but with partial overlap as in mode 4;
6. same as mode 3, but with partial overlap as in mode 4.

We performed 14 runs for each round of experiments. The log generation modes, noise frequency, and overlap percentage used are reported in Figure 3.10. Note that default values are log generation mode 6 (that is the most complex mode, which also captures the fact that noise more often appears *between* alert symbols of actual interest), noise frequency 3/10, and overlap percentage 40%.[3] We assumed worst-case behavior of function γ, i.e., it always returns *true*.

We also performed experiments with much larger logs (1M tuples) – interestingly, the performance we obtained in terms of tuples processed per second was 5.1% worse at most.

Run	**Log gen. mode**	**Noise frequency**	**Overlap percentage**
1	1	–	–
2	2	3/10	–
3	3	3/10	–
4	4	–	40%
5	5	3/10	40%
6	6	3/10	40%
7	6	1/10	40%
8	6	2/10	40%
9	6	3/10	40%
10	6	4/10	40%
11	6	5/10	40%
12	6	3/10	20%
13	6	3/10	30%
14	6	3/10	40%
15	6	3/10	50%
16	6	3/10	60%

Fig. 2.11. Parameter values used for each experimental run.

2.4.2 Results

Figure 2.12 reports the results of the first round of experiments. In particular, Figure 2.12(top) shows the number of log tuples processed per second when varying the log generation mode (runs 1–6), Figure 2.12(center) shows the

[3] For simplicity of presentation, the run with all parameters set to default values is reported as three separate runs (6, 9, and 14) in Figure 3.10.

variation with respect to noise frequency (runs 7–11), and Figure 2.12(bottom) the variation with respect to the percentage of overlap between consecutive occurrences in the log (runs 12-16).

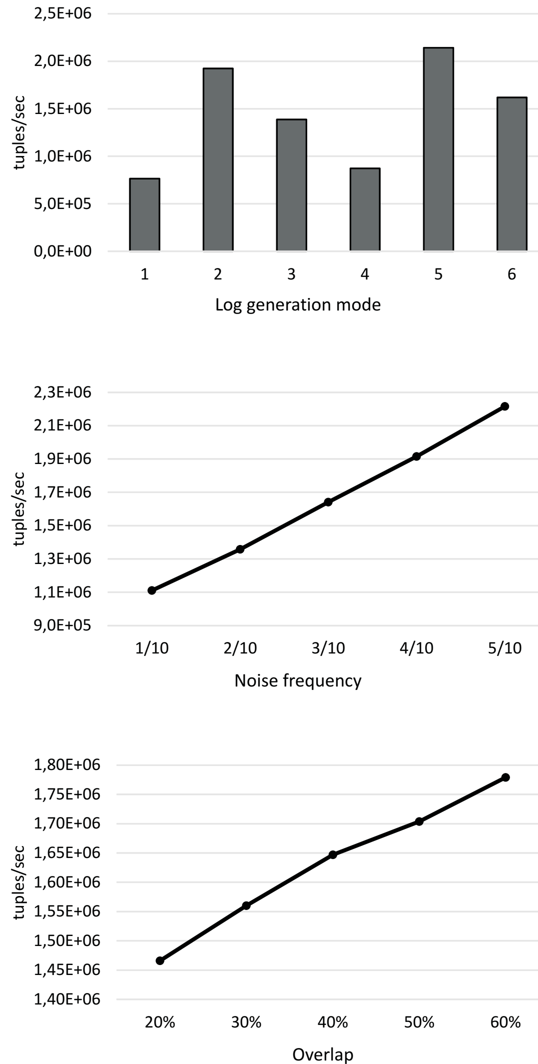


Fig. 2.12. Tuple rates in the first round of experiments.

The results confirm our expectations and show extremely good overall performances. In Figure 2.12(top) we can notice that, as expected, the presence of noise in the log or overlap between consecutive instances reduces the overall

number of occurrences, thus improving performances. Moreover, when noise appears instead of alert symbols of actual interest (which we believe is an even more realistic case), we obtain better results than when noise appears between such symbols. Generally, the number of tuples processed per second is extremely high – it is consistently higher than 765K, and 1.4M on average. In both Figure 2.12(center) and Figure 2.12(bottom) the trend is basically linear in the frequency of noise and percentage of overlap – in these experiments, the average tuple rate is around 1.6M tuples/sec.

The second round of experiment used patterns P_5 and P_6 to outline the behavior of our framework when varying the "density" of the patterns, i.e., the number of edges w.r.t. the number of vertices – much denser patterns usually yield a much bigger AC-Index as each log tuple can be attached to many more occurrences.

Figure 2.13 shows the tuple rates obtained in this round. Again, the results appear very satisfactory. We can notice in Figure 2.13(top) that the performance loss is always around 40% when moving from a sparse pattern (P_6) to a much denser one (P_5). The tuple rate never dropped below 260K tuples/sec, and it was around 700K tuples/sec on average. In the experiments where we fixed the log generation mode to 6 and varied noise frequency and overlap percentage (center and bottom of the figure) the performance loss was always around 60%. It should be observed that the number of paths in P_5 is 64 times that of P_6. Thus, the relationship between the number of paths and the tuple rates is much less than linear.

We also measured the number of occurrences and the indexing time per tuple normalized by the number of occurrences. The results for all the runs of the second round are reported in Figure 2.14. As expected, the number of occurrences is lower when using P_6. Interestingly, the normalized indexing time shows very small variations with respect to the specific configuration used (8% on average), and the framework appears to use slightly more resources per occurrence when using P_6.

Finally, Figure 2.15 reports the maximum size and the normalized maximum size of the AC-Index for all the runs of the second round. As expected, using P_5 produces a much larger AC-Index – the difference was around 60% on average (again, showing a sub-linear relationship with the number of paths in the patterns). Moreover, in this case the size of the AC-Index shows very small variations with respect to the configuration used. Again, the framework appears to use slightly more resources per occurrence when using P_6.

Wrapping up, we can draw the following general conclusions from the experimental results:

- Overall, the framework is able to process logs that enter the system at extremely large rates – orders of magnitude of 100K–1M tuples/sec are definitely sufficient for fully covering a wide range of real-world applications.

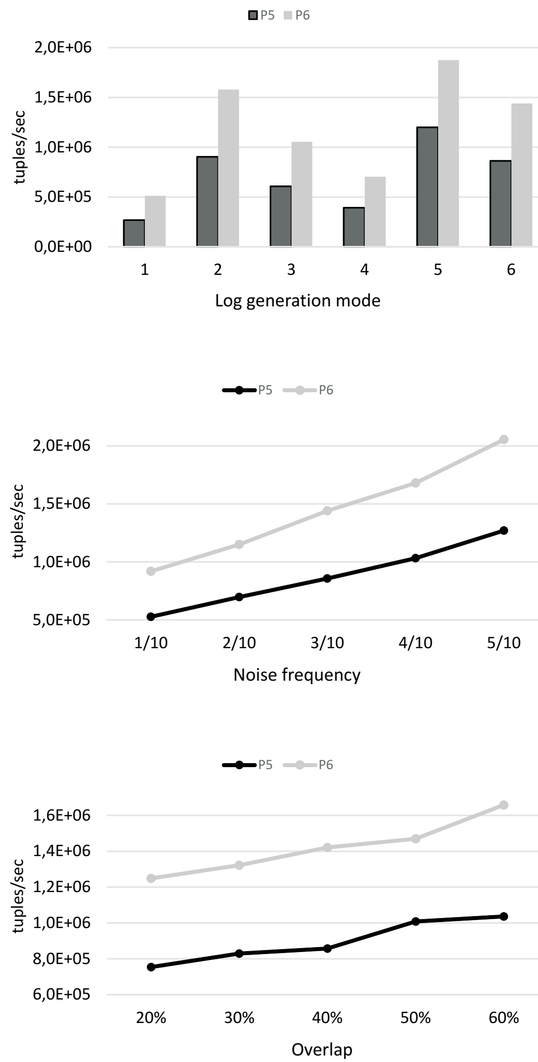


Fig. 2.13. Tuple rates in the second round of experiments.

- The framework scales well w.r.t. the amount of noise in the log and overlap between consecutive occurrences.
- When working with much denser patterns, the framework scales well with the much higher number of occurrences retrieved. It shows good “stability” of the normalized indexing time per tuple and of the size of the AC-Index w.r.t. the various experimental parameters used. Interestingly, the

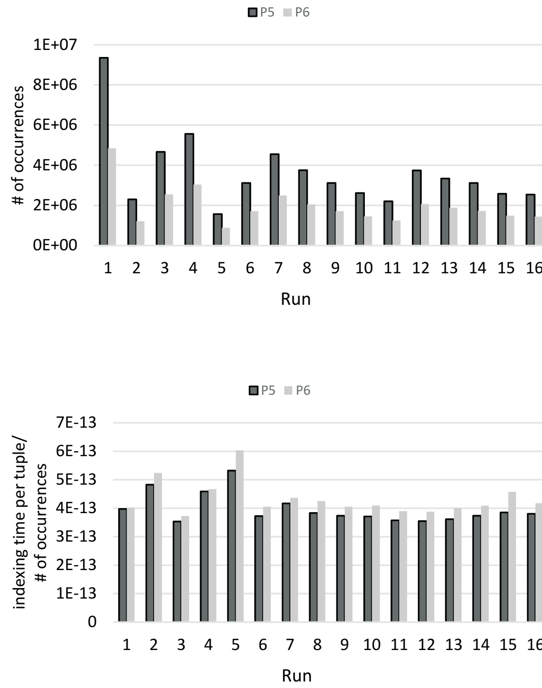


Fig. 2.14. Number of occurrences (top) and normalized indexing time per tuple (bottom) in the second round of experiments.

"efficiency per occurrence" of the framework is slightly higher with denser patterns.

2.5 Related Works

2.5.1 Graph-Based Alert Correlation

A number of interesting graph-based alert correlation techniques has been proposed in the past. Attack graphs and finite automata have often been used for this purpose. In [27] Michael et al. proposed a technique for identifying malicious execution traces with automatically-learned finite automata. Sekar et al. [28] created an automaton-based approach for detecting anomalous program behaviors. Each node in the DFA represents a state in the program under inspection which the algorithm utilizes to learn "normal" data and perform detection. [29] proposes to increase the accuracy of the N-gram learning algorithm by using a DFA representation for intrusion detection via system call traces. In [30] Wagner and Dean show how static analysis may

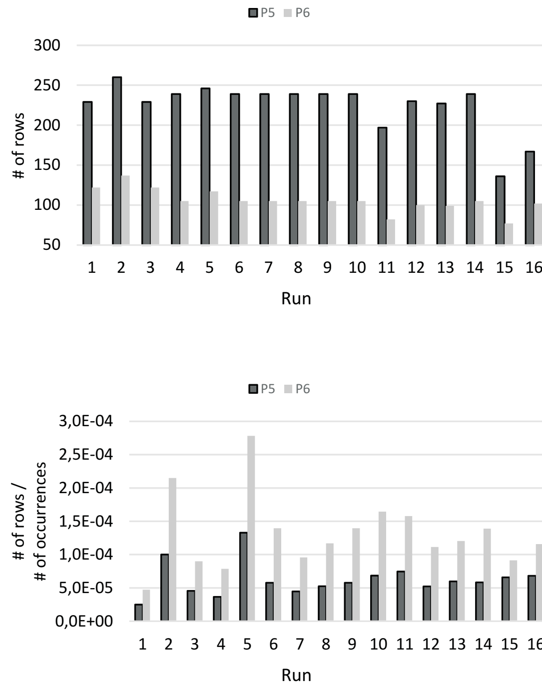


Fig. 2.15. Maximum size (top) and normalized maximum size (bottom) of the AC-Index in the second round of experiments.

be used to automatically derive a model of application behavior for intrusion detection. They make use of a nondeterministic DFA to characterize the expected system call traces. In [1] a hypergraph-based intrusion detection model correlates groups of alerts that may occur in any order. In [31] a technique is presented to automatically produce candidate interpretations of detected failures from anomalies identified by detection techniques that use inferred DFAs to represent the expected behavior of software systems. In [32] Branch et al. describe an approach for the real-time detection of denial of service attacks using time-dependent DFAs. They use the time intervals between certain event occurrences, as defined in a DFA, to improve the accuracy of detecting specific attacks. [23] proposes a correlation algorithm based on attack graphs that is capable of detecting multiple attack scenarios for forensic analysis. In [22] attack graphs are used for correlating, hypothesizing, and predicting intrusion alerts. [24] proposes to represent groups of alerts with graph structures, along with a method that automatically identifies frequent groups of alerts and summarizes them into a suspicious sequence of activity. In [33] a framework is described for managing network attack graph complexity through interactive visualization, which includes hierarchical aggregation

of graph elements. [25] proposes an automated technique for generating and analyzing attack graphs, based on symbolic model checking algorithms. [6, 34] construct attack scenarios that correlate critical events on the basis of prerequisites and consequences of attacks. [35] focuses on the online approach to alert correlation by employing a Bayesian network to automatically extract information about the constraints and causal relationships among alerts. Finally, [36] introduces a host-based anomaly intrusion detection methodology using discontinuous system call patterns.

2.5.2 Fusion-Based Alert Correlation

Fusion-based correlation techniques make use of correlation functions in order to store, map, cluster, merge, and correlate alerts. [37] proposes a multisensor data fusion approach for intrusion detection. Cuppens [38, 39] suggests to design functions which recognize alerts corresponding to the same occurrence of an attack and create a new alert that merges data contained in those alerts. [40] presents a probabilistic approach to alert correlation by extending ideas from multisensor data fusion. Their fusion algorithm only considers common features in the alerts to be correlated, and for each feature they define an appropriate similarity function.

2.5.3 Pattern Discovery

Sun et al. proposed several methods for pattern discovery. In [41] a distributed hierarchical real-time algorithm is described that works with group of streams. Patterns discovered from each stream are merged in order to obtain global patterns across groups. [42] provides a mining technique based on tensor analysis. Timestamp, type, and location attruibutes of data stream values are modeled using a tensor, giving particular attention to the temporal aspect. In [43] a pattern discovery technique for streaming applications is proposed. The tensor stream is a general dynamic data model successfully used for this purpose, where data streams and time-evolving graphs become the rst and second order special cases.

2.5.4 Other Works

In [44] the main objective is to analyze the alert correlation techniques that are able to improve the intrusion detection task in terms of alert flooding, false alerts and scalability in heterogeneous log scenarios. [45] describes an aggregation and correlation algorithm for acquiring intrusion detection alerts and relating them together to expose a more condensed view of the security issues raised. In [20] Valeur et al. propose a general correlation model, which provides cascaded stages to normalize, pre-process, fuse, verify, and correlate alerts and reconstruct attacks.

2.6 Conclusions

In this chapter we proposed a technique for alert correlation based on DFA-like patterns and user-provided correlation functions that supports the fast retrieval of occurrences of the given patterns through appropriate indexing and retrieval schemes. The experimental results have proven that, although the proposed model is very expressive, the specifically-designed indexing and retrieval schemes are able to guarantee a very high efficiency of the retrieval process.

3

Intrusion Detection with Hypergraph-Based Attack Models

In numerous security scenarios, given a sequence of logged actions, it is necessary to look for all subsequences that represent an intrusion, which can be meant as any "improper" use of a system, an attempt to damage parts of it, to gather protected information, to follow "paths" that do not comply with security rules, etc. In this Chapter we propose a hypergraph-based attack model for intrusion detection. The model allows the specification of various kinds of constraints on possible attacks and provides a high degree of flexibility in representing many different security scenarios. We discuss the main features of the model and study the problems of checking the consistency of attack models and detecting attack instances in sequences of logged actions. Moreover, we propose an index structure and its associated maintenance and retrieval algorithms, which are designed to concurrently track instances of multiple attack models. The efficiency of our proposed index and algorithms is confirmed by an extensive experimental evaluation.

3.1 Introduction

In numerous security scenarios, given a sequence of logged actions, it is necessary to look for all subsequences that represent an intrusion. The *intrusion detection* task is not generally restricted to the field of computer networks: for instance, it includes the scenarios where a surveillance system is active over a public area. An intrusion can thus be meant as any "improper" use of a system, an attempt to damage parts of it, to gather protected information, to follow "paths" that do not comply with security rules, etc.

In this Chapter we propose a hypergraph-based attack model for intrusion detection. Our proposed model is capable of:

- representing many different attack types/structures in a compact way;
- expressing temporal constraints on the execution of attacks;
- accommodating many different security scenarios;

- representing attack scenarios at different abstraction levels, allowing to "focus" the intrusion detection task in various ways.

An attack model is defined in such a way that the paths from start to terminal hyperedges in the model represent an attack and correspond to subsequences of a given input log—we assume that the log is a sequence of events (tuples) having a type and a timestamp. Such subsequences are the *instances* of the attack model.[1] A group of log tuples of the same type form the instance of a vertex, whereas the associations among groups form the instance of a hyperedge (called *segment*). The model allows expressing various kinds of constraints: vertices can specify type and cardinality constraints on groups of tuples and hyperedges can specify temporal constraints on the associations among groups.

It should be observed that our notion of attack model assumes that *all possible attacks* follow a path from a start to a terminal hyperedge in the model. As a matter of fact, in numerous security applications, attack models cover all possible attacks since they are defined on the basis of the specific network configuration of an organization. Actions in the model basically correspond to interdependent security vulnerabilities/alerts on specific machines – attackers *cannot* follow different paths because this would require, e.g., traversing firewalled network sections [6].

The following example intuitively introduces the main features of the proposed model.

Example 3.1. Consider the attack model in Fig. 3.1, where actions are logged security alerts and are depicted with plain circles (v_i), while hyperedges are depicted with dotted circles (h_i).

In this attack model:

- h_1 is a start hyperedge (indicated with a white arrow) so an attack can begin with it. Vertex v_1, labeled with Scan, requires the presence of exactly one log tuple of type Scan (cardinality constraint "(1:1)");
- hyperedge h_2 represents an association between vertices v_1, v_2, and v_3, requiring that the corresponding instance (log segment) contains exactly one Scan tuple along with optional Local Access and Remote Access tuples (cardinality constraint "(0:1)"), in any order. Moreover, there is a temporal constraint $(3, 6)$ that mandates, for the log segment, a temporal extension between 3 and 6 time points (note that this actually ensures that at least one Local Access or Remote Access tuple is present in the segment);
- hyperedge h_3 represents an association between vertices v_2, v_3, and v_4, indicating that an instance of h_2 can be extended by one or more Directory Traversal tuples (cardinality constraint "(1:∞)");
- the same applies to hyperedges h_4 and h_5;

[1] Note that, although we use the word "instance", the input logs are not assumed to be generated according to our model.

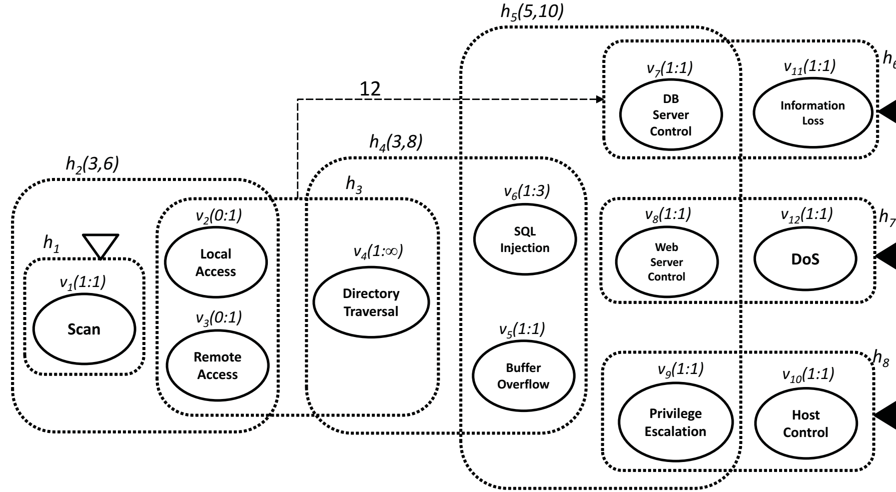


Fig. 3.1. Example attack model

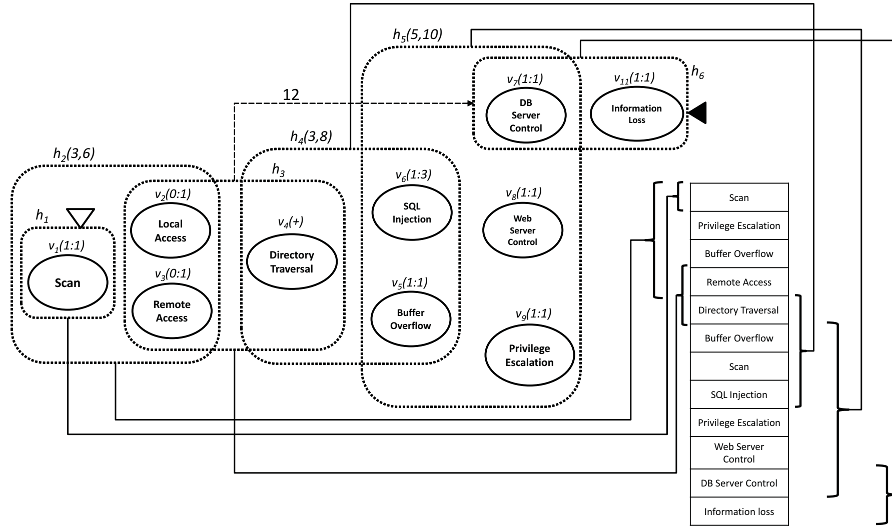
- h_6 is a terminal hyperedge (indicated with a black arrow), so an attack can end with it.
- the edge between h_3 and h_6 adds a further temporal constraint: the log tuple where the segment which is an instance of h_3 starts must appear at most 12 time points before the tuple where the segment for h_6 starts.

If we consider the log in Fig. 3.2(a), the corresponding instance is graphically described in Fig. 3.2(b). Further details will be provided in the next section.

The remainder of the paper is organized as follows. In Section 3.2 we give the formal definitions of attack models and instances. In Section 3.3 we characterize the problem of consistency checking for attack models and give related theoretical results. In Section 3.4 we formally define the intrusion detection problem we are interested in, and characterize its complexity. In Section 3.5 we introduce the generalization/specialization of actions through is-a relationships and briefly discuss their applications and related issues. In Section 3.6 we propose an index data structure and its associated maintenance and retrieval algorithms that index a log with respect to multiple attack models in order to quickly find attack instances in it. In Section 4.7 we show experimental results which confirm the validity of our index and algorithms. Finally, Section 3.8 discusses related work and Section 3.9 outlines conclusions.

Log tuple	Type	Timestamp
ℓ_1	Scan	0
ℓ_2	Privilege Escalation	1
ℓ_3	Buffer Overflow	2
ℓ_4	Remote Access	5
ℓ_5	Directory Traversal	7
ℓ_6	Buffer Overflow	10
ℓ_7	Scan	11
ℓ_8	SQL Injection	12
ℓ_9	Privilege Escalation	14
ℓ_{10}	Web Server Control	15
ℓ_{11}	DB Server Control	17
ℓ_{12}	Information Loss	20

(a)



(b)

Fig. 3.2. Example log (a) and instance of the model (b)

3.2 Modeling Attack Processes

In this section we give the formal definitions of our proposed attack model and its instances. We assume that an alphabet $\mathcal{A}$ of symbols is given, univocally identifying the action types of the underlying process of attack.

Definition 3.2 (Attack Model). *An* attack model *defined over the set of actions $\mathcal{A}$ is a tuple $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ where:*

- *$\mathcal{H} = (V, H)$ is a hypergraph, where V is a finite set of vertices and H is a set of hyperedges (i.e., for each $h \in H$, $h \subseteq V$).*

- $\lambda : V \to \mathcal{A} \times \mathbb{N}^0 \times (\mathbb{N}^+ \cup \{\infty\})$ *is a vertex labeling function that associates with each vertex* $v \in V$ *a triple of the form* (a, l, u)*, with* $l \leq u$*, which specifies the action of* v *along with its cardinality constraints.*[2]
- $\tau : H \to \mathbb{N}^0 \times (\mathbb{N}^+ \cup \{\infty\})$ *is a (partial) function that expresses temporal constraints (in terms of lower and upper bounds) on hyperedges—*$domain(\tau)$ *will denote the set of hyperedges* $h \in H$ *such that* $\tau(h)$ *is defined;*
- $\epsilon : H \times H \to \mathbb{N}^+$ *is a (partial) function that express temporal constraints (in terms of upper bounds) on ordered pairs of distinct hyperedges—*$domain(\epsilon)$ *will denote the set of pairs of hyperedges* $h_i, h_j \in H$ *such that* $\epsilon(h_i, h_j)$ *is defined;*
- $S, T \subseteq H$ *are nonempty sets of* start *and* terminal *hyperedges, respectively.*

The following example shows how the model of Fig. 3.1 is formalized according to Definition 3.2.

Example 3.3. In the attack model $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ of Fig. 3.1 we have:

- $V = \{v_1, \ldots, v_{12}\}$;
- $H = \{h_1 = \{v_1\}, h_2 = \{v_1, v_2, v_3\}, h_3 = \{v_2, v_3, v_4\}, h_4 = \{v_4, v_5, v_6\}, \ldots\}$;
- $\lambda(v_1) = (\textsf{Scan}, 1, 1)$, $\lambda(v_2) = (\textsf{Local Access}, 0, 1)$, $\lambda(v_3) = (\textsf{Remote Access}, 0, 1)$, $\lambda(v_4) = (\textsf{Directory Traversal}, 1, \infty)$, $\lambda(v_5) = (\textsf{Buffer Overflow}, 1, 1)$, etc.;
- $domain(\tau) = \{h_2, h_4, h_5\}$, $\tau(h_2) = (3, 6)$, $\tau(h_4) = (3, 8)$, $\tau(h_5) = (5, 10)$;
- $domain(\epsilon) = \{(h_3, h_6)\}$, $\epsilon(h_, h_6) = 12$;
- $S = \{h_1\}$, $T = \{h_6, h_7, h_8\}$.

We now define paths in an attack model.

Definition 3.4 (path). *A* path π *in an attack model* $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ *is a sequence* $h_1, \ldots, h_m$ *of distinct hyperedges from* $\mathcal{H}$ *such that*

1. $h_1 \in S$;
2. $\forall i \in \{2, ..., m\}$, $h_{i-1} \cap h_i \neq \emptyset$;
3. *there is no index* $j \in \{2, ..., m-1\}$ *such that* $h_1, ..., h_{j-1}, h_{j+1}, ..., h_m$ *satisfies both Conditions 1 and 2.*

Moreover, π *is said to be* complete *if* $h_m \in T$*.*

A *log* is a sequence $\ell_1, ..., \ell_n$, with $n > 0$ and where each ℓ_i is a tuple $\langle att_1, \ldots, att_k \rangle$ of attributes (e.g., user-id, IP, etc.). In the following, we assume that a '*timestamp*' attribute, here just formalized as a natural number, encodes the time point (w.r.t. an arbitrary but fixed time granularity) at which the action represented by a log tuple occurs. Moreover, for each $i, j \in \{1, ..., n\}$

[2] The intended meaning of the symbol '∞' is that there is no upper bound.

with $i < j$, it holds that $\ell_i.timestamp < \ell_j.timestamp$, i.e., the sequence reflects the temporal ordering of the tuples.[3] Moreover, we assume that a '*type*' attribute encodes the action.

Before defining the instance of an attack model, we introduce the notion of *m-segmentation.*

Definition 3.5 (m-segmentation). *Let $L = \ell_1, ..., \ell_n$ be a log and let $m > 0$ be a natural number. A* segment *of L is a pair (s, t) of natural numbers such that $1 \leq s \leq t \leq n$. An m-*segmentation *of L is a sequence $(1 = s_1, t_1), ..., (s_m, t_m = n)$ of pairs of natural numbers such that:*

1. *$\forall i \in \{1, ..., m\}$, (s_i, t_i) is a segment of L;*
2. *$\forall i \in \{1, ..., m-1\}$, $s_{i+1} \leq t_i$.*

Example 3.6. Consider the log L of Fig. 3.2(a)—for the moment, ignore the timestamps. The sequence $(1, 1), (1, 4), (4, 5), (5, 8), (6, 11), (11, 12)$ is a 6-segmentation of L that segments it into the sub-logs $L_1 = \ell_1$, $L_2 = \ell_1, \ldots, \ell_4$, $L_3 = \ell_4, \ldots, \ell_5$, $L_4 = \ell_5, \ldots, \ell_8$, $L_5 = \ell_6, \ldots, \ell_{11}$, and $L_6 = \ell_{11}, \ldots, \ell_{12}$.

Finally, given a log $L = \ell_1, \ldots, \ell_n$, we define the temporal distance between two tuples ℓ_i and ℓ_j in L as $d(\ell_i, \ell_j) = |\ell_j.timestamp - \ell_i.timestamp|$. We are now ready to formalize the notion of *instance* of an attack model.

An *instance* of an attack model M over a complete path in M is a log that can be segmented in such a way that, for each segment:

1. the types of the log tuples in the segment comply with the corresponding hyperedge;
2. the segment does not include unnecessary tuples as its start or end;
3. the temporal extension of the segment complies with the constraints specified by function τ (if present);
4. the start tuples of two consecutive segments comply with the constraints specified by function ϵ (if present).

The following definition formalizes this.

Definition 3.7 (Instance of an Attack Model). *Assume that $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ is an attack model over $\mathcal{A}$. Let $\pi = h'_1, \ldots, h'_m$ be a complete path in M, and let $L = \ell_1, \ldots, , \ell_n$ be a log. Then, we say that L is an* instance of M over π, *denoted by $L \models_\pi M$, if there exists an m-segmentation $(s_1, t_1), ..., (s_m, t_m)$ of L such that $\forall i \in \{1, \ldots, m\}$,*

1. *$\forall\, v \in h'_i$, if $\lambda(v) = (a, l, u)$, then $l \leq |\{\ell \in \ell_{s_i}, ..., \ell_{t_i} | \ell.type = a\}| \leq u$;*
2. *$\exists\, v_s, v_t \in h'_i$ such that $\lambda(v_s) = (\ell_{s_i}.type, l_s, u_s)$ and $\lambda(v_t) = (\ell_{t_i}.type, l_t, u_t)$;*
3. *if $h'_i \in domain(\tau)$, then $l_i \leq d(\ell_{s_i}, \ell_{t_i}) \leq u_i$, with $\tau(h'_i) = (l_i, u_i)$;*

[3] Note that we are assuming here, w.l.o.g., that there are no tuples with the same timestamp. Indeed, this can always be guaranteed by assuming a sufficiently fine time granularity.

4. $\forall i, j \in [1, m]$ *s.t.* $i < j$ *and* $(h'_i, h'_j) \in domain(\epsilon)$*, it holds that* $d(\ell_{s_i}, \ell_{s_j}) \leq \epsilon(h'_i, h'_j)$.

Example 3.8. Consider the attack model M of Fig. 3.1 and the path $\pi = h_1, h_2, h_3, h_4, h_5, h_6$ in M. The log L in Fig. 3.2(a) is an instance of M over π because the 6-segmentation $(1,1), (1,4), (4,5), (5,8), (6,11), (11,12)$ of L is such that (see Fig. 3.2(b)):

- the sets of action types associated with sub-logs $L_1 = \ell_1$, $L_2 = \ell_1, \ldots, \ell_4$, $L_3 = \ell_4, \ldots, \ell_5$, $L_4 = \ell_5, \ldots, \ell_8$, $L_5 = \ell_6, \ldots, \ell_{11}$, and $L_6 = \ell_{11}, \ldots, \ell_{12}$, are "minimal" supersets of the sets of action types associated with hyperedges h_1, h_2, h_3, h_4, h_5 and h_6, respectively (Conditions 1 and 2 in Definition 3.7);
- temporal constraints hold (Conditions 3 and 4 in Definition 3.7):
  - $\tau(h_2) = (3,6)$ and $3 \leq d(\ell_1, \ell_4) = 5 \leq 6$;
  - $\tau(h_4) = (3,8)$ and $6 \leq d(\ell_5, \ell_8) = 5 \leq 8$;
  - $\tau(h_5) = (5,10)$ and $5 \leq d(\ell_6, \ell_{11}) = 7 \leq 8$;
  - $\epsilon(h_3, h_6) = 12$ and $d(\ell_1, \ell_4) = 9 \leq 12$.

3.3 Consistency of Attack Models

In this section we study the consistency of attack models and the complexity of checking whether a given attack model is consistent. We start by defining the consistency of a path.

Definition 3.9 (Consistency of a path). *Let* π *be a complete path in an attack model* $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$*. We say that* π *is* consistent *w.r.t.* M *if there is an instance* L *of* M *over* π*, i.e., if there is a log* L *such that* $L \models_\pi M$*.*

To detect the consistency of a complete path $\pi = h'_1, \ldots, h'_m$ in M, we associate a *support graph* with π, denoted by $SG(M, \pi) = \langle N, E, \omega \rangle$, that is a node- and edge-weighted directed graph where:

- the set N of nodes are the hyperedges of π;
- there is precisely an edge from h'_α to h'_β in E for each $(h'_\alpha, h'_\beta) \in domain(\epsilon)$ with $\alpha < \beta$;
- $\omega(h'_i) = \begin{cases} l_i & \text{if } h'_i \in domain(\tau) \text{ and } \tau(h'_i) = (l_i, u_i); \\ 0 & \text{if } h'_i \notin domain(\tau); \end{cases}$
- $\omega(h'_\alpha, h'_\beta) = \epsilon(h'_\alpha, h'_\beta)$.

Example 3.10. Consider the attack model M' obtained from the model M of Fig. 3.1 by adding the temporal constraint $\epsilon(h_2, h_5) = 5$. The graph $SG(M, \pi = h_1, h_2, h_3, h_4, h_5, h_6)$ is shown in Fig. 3.3. By definition, the graph has 6 nodes/hyperedges, and two edges corresponding to the two elements in the domain of ϵ.

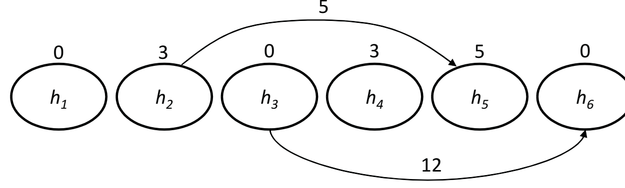


Fig. 3.3. Support graph associated with the path $h_1, h_2, h_3, h_4, h_5, h_6$

The following result gives us necessary and sufficient conditions for a path to be consistent.

Theorem 3.11. *Let M be an attack model and let $\pi = h'_1, \ldots, h'_m$ be a complete path in M. Then, π is consistent w.r.t. M if and only if for each edge (h'_α, h'_β) in $SG(M, \pi)$, it holds that*

$$\sum_{i=\alpha}^{\beta-1} \omega(h'_i) \leq \omega(h'_\alpha, h'_\beta).$$

For instance, in Fig. 3.3, we have that $\epsilon(h_2, h_5) = 5 < w_{SG}(h_2)+w_{SG}(h_3)+w_{SG}(h_4) = 6$. Thus, by Theorem 3.11, π is not consistent w.r.t. M'.

We now define three kinds of consistency notions for attack models.

Definition 3.12 (Consistency of an attack model). *Assume that $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ is an attack model. We say that:*

1. *M is (S/T)-consistent if $\forall h_s \in S$, $\forall h_m \in T$ there is a path π starting with h_s and ending with h_m, respectively, that is consistent;*
2. *M is (S)-consistent if $\forall h_s \in S$, there is a complete path π starting with h_s that is consistent;*
3. *M is (T)-consistent if $\forall h_m \in T$, there is a path π terminating with h_m that is consistent.*

Observation 1 *Let M be an attack model. If M is (S/T)-consistent, then M is both S-consistent and T-consistent.*

Finally, we characterize the complexity of checking the consistency of an attack model.

Theorem 3.13. *Deciding whether a given attack model is (S/T)-consistent is* **NP***-complete, even if $|S| = |T| = 1$.*

Proof. Membership in **NP** is trivial. For the hardness, we give a reduction from the Monotone one-in-three 3SAT problem, which is known to be **NP**-complete [46]. The problem is a variant of the classical satisfiability problem, where the input instance is a conjunction of clauses, with each clause consisting of exactly three variables (i.e., negation is not allowed). The goal is

to determine whether there is a truth assignment to the variables so that each clause has exactly one true variable (and thus exactly two false variables).

Let $\phi = C_1 \wedge ... \wedge C_m$ be a Boolean formula taken as input and such that each clause $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}$, $\forall i \in \{1, ..., m\}$, consists of exactly 3 variables. Based on ϕ, we define an attack model $M(\phi) = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ over a set $\mathcal{A}$ of actions, where $\mathcal{H} = (V, H)$ and such that:

1. $V = \{s, \phi, t, \bar{\phi}\} \cup \{C_i, x_{i,j}, \bar{C}_i, \bar{x}_{i,j} \mid i \in \{1, ..., m\} \wedge j \in \{1, 2, 3\}\}$;
2. the set H exactly contains the following hyperedges:
   - $h_s = \{s, C_1\}$;
   - $h_{i,j} = \{C_i, x_{i,j}\}$ and $h^{\wedge}_{i,j} = \{x_{i,j}, C_{i+1}\}$, $\forall i \in \{1, ..., m-1\} \wedge j \in \{1, 2, 3\}$;
   - $h_{m,j} = \{C_m, x_{m,j}\}$ and $h^{\wedge}_{m,j} = \{x_{m,j}, \phi\}$, $\forall j \in \{1, 2, 3\}$;
   - $\bar{h}_s = \{\phi, \bar{C}_1\}$;
   - $\bar{h}_{i,j} = \{\bar{C}_i, \bar{x}_{i,j}\}$ and $\bar{h}^{\wedge}_{i,j} = \{\bar{x}_{i,j}, \bar{C}_{i+1}\}$, $\forall i \in \{1, ..., m-1\} \wedge j \in \{1, 2, 3\}$;
   - $\bar{h}_{m,j} = \{\bar{C}_m, \bar{x}_{m,j}\}$ and $\bar{h}^{\wedge}_{m,j} = \{\bar{x}_{m,j}, \bar{\phi}\}$, $\forall j \in \{1, 2, 3\}$; and,
   - $h_t = \{\bar{\phi}, t\}$;
3. $\lambda(v) = (v, 1, 1)$, for each $v \in V$; in fact, $\mathcal{A} = V$;
4. $domain(\tau) = \{\bar{h}_s\}$, and $\tau(\bar{h}_s) = (2, 3)$;
5. $S = \{h_s\}$, $T = \{h_t\}$;
6. for each variable of the form $x_{i,j}$ with $i \in \{1, ..., m\}$ and $j \in \{1, 2, 3\}$, for each clause C_z with $z \in \{1, ..., m\}$ where $x_{i,j}$ occurs (possibly with $i = z$), and for each variable $x_{z,k} \neq x_{i,j}$ with $k \in \{1, 2, 3\}$ occurring in C_z and different from $x_{i,j}$, we have that: $(h_{i,j}, \bar{h}_{z,k}) \in domain(\epsilon)$ and $\epsilon(h_{i,j}, \bar{h}_{z,k}) = 1$.

As an example, the formula $\phi = (x \vee y \vee z) \wedge (x \vee y \vee w)$ is a YES instance to the MONOTONE ONE-IN-TREE 3SAT problem, as it witnessed by the truth assignment where x is the only variable evaluating true. The hypergraph associated with the attack model $M(\phi)$ is shown in Fig. 3.4, where for the sake of readability only arrows associated with constraints involving the variable x in ϕ are depicted.

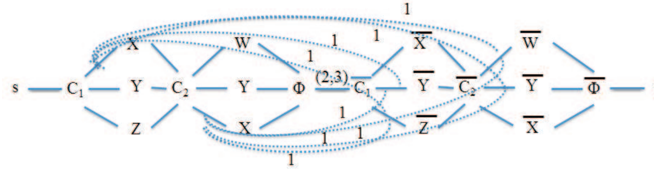


Fig. 3.4. The graph reduction constructed from ϕ.

Now, we complete the proof by claiming that: *ϕ is a YES instance to the* Monotone one-in-three 3SAT *problem $\Leftrightarrow$ there $M(\phi)$ is (S/T)-consistent.*

($\Rightarrow$) Let θ be an assignment witnessing that ϕ is a YES instance. Based on θ, we build the sequence of vertices $v(\theta) = s, C_1, x_{1,j_1}, C_2, x_{2,j_2}, ..., C_m, x_{m,j_m}, \phi, \bar{C}_1, \bar{x}_{1,j_1}, \bar{C}_2, \bar{x}_{2,j_2}, ..., \bar{C}_m, \bar{x}_{m,j_m}, \bar{\phi}, t$ where x_{i,j_i} is the only variable evaluating true w.r.t. θ in the clause C_i, $\forall i \in \{1, ..., m\}$. Moreover, we build the sequence $\pi(\theta) = \{v_1, v_2\}, ..., \{v_i, v_{i+1}\}, ..., \{v_{4m+3}, v_{4m+4}\}$, where v_i is the i-th vertex in the sequence $v(\theta)$. Note that $\pi(\theta)$ is a path in $M(\phi)$ starting with h_s and terminating with h_t. In addition, it is consistent. Indeed, let $L_\theta = l_1, l_2, ..., l_{4m+4}$ be a log such that $l_i.type = v_i$ and $l_i.timestamp = i$, for each $i \in \{1, ..., 4m+4\}$. Then, just notice that $L \models_{\pi(\theta)} M(\phi)$.

($\Leftarrow$) Let π be a consistent path in $M(\phi)$ starting with h_s and terminating with h_t. Note that by definition of a path, π must be of the following form: $\pi = h_s, h_{1,a_1}, h^{\wedge}_{1,a_1}, ..., h_{m,a_m}, h^{\wedge}_{m,a_m}, \bar{h}_s, \bar{h}_{1\bar{a}_1}, \bar{h}^{\wedge}_{1,\bar{a}_1}, ..., \bar{h}_{m,\bar{a}_m}, \bar{h}^{\wedge}_{m,\bar{a}_m}, h_t$, where $a_1, ..., a_m, \bar{a}_1, ..., \bar{a}_m \in \{1, 2, 3\}$. Now, recall that $\tau(\bar{h}_s) = (2, 3)$, i.e., in particular, the duration of $\bar{h}_s$ is at least 2 time units. Given the construction of ϵ, it follows that π cannot contain any pair of hyperedges in the domain of ϵ. Therefore, for each hyperedge h_{i,a_i}, with $i \in \{1, ..., m\}$, if the variable x_{i,a_i} occurs in the clause C_z, with $z \in \{1, ..., m\}$, then the hyperedge $\bar{h}_{z,\bar{a}_z}$ is actually such that $x_{z,\bar{a}_z} = x_{i,a_i}$. Let now θ be the truth assignment such that the variable x_{i,a_i} evaluates true, for each $i \in \{1, ..., m\}$ (and the remaining variables evaluate false). Note that θ is satisfying. Then, assume by contradiction that there is a clause C_z and two distinct variables $x_{z,\alpha}$ and $x_{z,\beta}$ evaluating true in θ. By construction of θ, it must be the case that $x_{z,\alpha} = x_{i',a_{i'}}$ and $x_{z,\beta} = x_{i'',a_{i''}}$, where i' and i'' are two indices in $\{1, ..., m\}$. However, by the above observations, $\bar{h}_{z,\bar{a}_z}$ is actually such that $x_{z,\bar{a}_z} = x_{i',a_{i'}} = x_{i'',a_{i''}}$. That is, $i' = i''$, which is impossible if the variables $x_{z,\alpha}$ and $x_{z,\beta}$ are distinct.

Interestingly, as the above result is obtained even when $|S| = |T| = 1$, the following result is entailed.

Corollary 3.14. *Deciding whether a given attack model is (S)-consistent (resp., (T)-consistent) is* **NP***-complete.*

Present research is investigating conditions under which consistency checking becomes tractable. In fact, tractable cases may arise when there are bounds on the number and structure of hyperedges and on the size of $domain(\epsilon)$.

3.4 The Intrusion Detection Problem

In this section we formally characterize the intrusion detection problem we are interested in and its complexity. The problem is basically that of checking

whether a log is an instance of an attack model. The following definition formalizes this.

Definition 3.15 (Intrusion Detection Problem). *Given an attack model M and a log L, determine whether there exists a complete path π in M such that $L \models_{\pi} M$.*

We now characterize the complexity of the above problem.

Theorem 3.16. *The intrusion detection problem is* **NP***-complete.*

Proof. Membership in **NP** is trivial: it suffices to use π and the corresponding $|\pi|$-segmentation as a polynomially-verifiable witness. We prove **NP**-hardness by polynomial-time reduction from HAMILTONIAN PATH [46]. Let $G_{in} = (V_{in}, E_{in})$ be an undirected graph. In order to decide whether G_{in} contains a Hamiltonian path, we build an attack model $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ as follows:

- $\mathcal{H} = (V, H)$;
- $V = V_{in}$;
- $H = \{\{v_1, v_2\} \mid (v_1, v_2) \in E_{in}\}$;
- $\forall v \in V$, $\lambda(v) = (x, 1, 1)$;
- $domain(\tau) = domain(\epsilon) = \emptyset$;
- $S = T = H$.

Then, we build a log $L = \ell_1, \ldots, \ell_n$ where $n = |V|$ and $\forall \ell_i \in L$, $\ell_i.type = x$ and $\ell_i.timestamp = i$. Now, in order to include all log tuples in an attack instance, all of the vertices in V must be traversed exactly once. Thus, there exists a complete path π in M such that $L \models_{\pi} M$ if and only if G_{in} contains a Hamiltonian path.

Theorem 3.16 establishes that there are cases where detecting an intrusion is not doable in polynomial time (unless **P**=**NP**). Intuitively, the exponential blowup is mainly due to the fact that, while analyzing the log:

1. it is necessary to maintain all possible “partial” instances (prefixes of the log that are “instances” over incomplete paths) of the attack model M;
2. many different incomplete paths in M can be associated with each partial instance;
3. many different segmentations of the partial instance can be associated with each (partial instance, incomplete path) pair.

Thus, when a new log tuple is analyzed and matched against the current set of partial solutions, many new partial solutions may be generated.

Recent works on the detection instances of temporal-automaton models in sequences of logged events [16, 6] have shown that acceptable detection times in real-world cases can be obtained by employing compact index structures and, most importantly, by limiting the number of partial solutions through

a form of early filtering based on temporal constraints. In other words, they have shown that the presence of temporal constraints allows to only look at the set of current partial solutions that lie within a fixed temporal "window". In Section 3.6 we define a new index structure and its associated detection algorithm that, based on the same intuition, allow our intrusion detection framework to obtain acceptable execution times – this will be confirmed by the experimental assessment described in Section 4.7.

3.5 Action Hierarchies

In this section we show how our proposed attack model can be seamlessly extended to handle the definition of generalization/specialization hierarchies among actions. The introduction of generalization/specialization relationships gives rise to two fundamental consequences:

- it allows to generalize/specialize an attack model when we want to track less (more) specific actions;
- it may affect the number of instances of the model and, as a consequence, the performance of the intrusion detection task.

Fig. 3.5(top) shows an example hierarchy for some of the software vulnerabilities used in the attack model of Fig. 3.3.

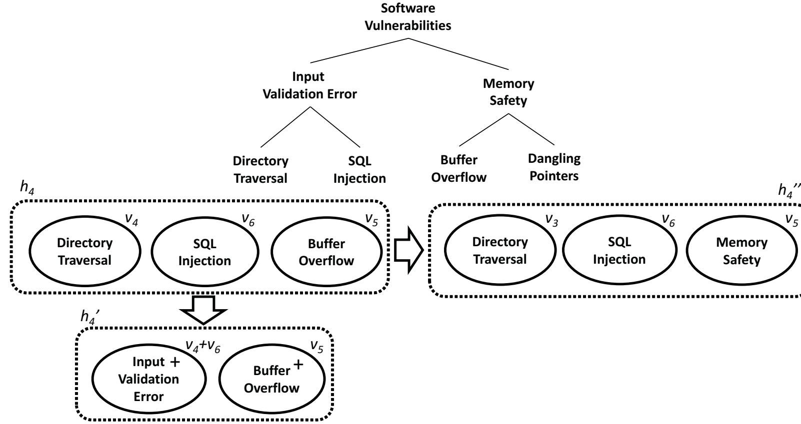


Fig. 3.5. Examples of abstraction based on action hierarchies

We model action hierarchies through an *is-a* relation $\Delta \subseteq \mathcal{A} \times \mathcal{A}$ and denote the transitive closure of Δ as Δ^+. With the introduction of generalization/specialization relationships, the following modifications have to be applied to Definition 3.2 and 3.7:

- In Definition 3.2, we add a requirement for function λ: it must be such that $\forall v, v' \in h$ with $\lambda(v) = (a, x, y)$ and $\lambda(v') = (a', x', y')$, it holds that $(a, a') \in \Delta^+$, $y' \geq x$. This ensures that the generalization/specialization relationships among the actions in h still allow the existence of log segments that are instances of h.
- In Definition 3.7, in order to correctly apply generalization/specialization relationships, Conditions 1 and 2 become:
  - $\forall v \in h'_i$, if $\lambda(v) = (a, l, u)$, then $l \leq |\{\ell \in \ell_s, ..., \ell_t | (\ell.type, a) \in \Delta^+\}| \leq u$;
  - $\exists\, v_s, v_t \in h'_i$ such that $\lambda(v_s) = (type_s, l_s, u_s)$, $\lambda(v_t) = (type_t, l_t, u_t)$, $(\ell_s.type, type_s) \in \Delta^+$, and $(\ell_t.type, type_t) \in \Delta^+$.

The following example shows how the use of action hierarchies allows us to easily generalize/specialize an attack model and how it affects the number of instances of the model.

Example 3.17. Suppose we define the action hierarchy of software vulnerabilities of Fig. 3.5(top) and we want to modify the attack model M of Fig. 3.1 for an intrusion detection task where it does not matter what kind of input validation error occurs. Fig. 3.5(bottom) shows how we could re-design hyperedge h_4 at a higher level of abstraction: v_4 and v_6 "collapse" into a single vertex, labeled with the action that is the closest common ancestor of the two. The resulting edge is h'_4. The attack model M' obtained this way is a generalization of M that better fits our needs. In this case, the number of instances of M' is likely to be larger than those of M, since we now also admit instances with neither SQL Injection nor Directory Traversal actions.

A larger number of instances can also be produced by generalizing h_4 through a re-labeling of v_5 with Memory Safety, thus producing h''_4. This way, the instances will admit Dangling Pointers log tuples as well.

3.6 Indexing and Detecting Attack Instances

In this section we present our proposed index structure, called AM-Index, and its associated maintenance and retrieval algorithms. As briefly discussed in Section 3.4, the index exploits as much as possible the temporal locality of the instances of the given attack models, along with their associated temporal constraints. The proposed index and algorithms are designed to be capable of concurrently tracking instances of multiple attack models. Moreover, besides retrieving the instances of any of the given models in the log, they return for each instance the path followed in the corresponding model.

In the remainder, given an attack model $M = \langle \mathcal{H}, \lambda, \tau, \epsilon, S, T \rangle$ and a hyperedge $h \in H$, we define $in_M(h) = \{h' \in H$ s.t. there exists a complete path π in M containing the subsequence $h', h\}$. Moreover, for the sake of conciseness, if $\lambda(v) = (a, l, u)$ we also write $label(v) = a$.

Definition 3.18 (AM-Index). *Given a set $\mathcal{M}$ of attack models and a log L, an AM-Index $I_{\mathcal{M}}$ is a pair $\langle \mathcal{T}, \mathcal{I} \rangle$ where:*

- *$\mathcal{T}$ is a set containing a table table(h) for each hyperedge present in $\mathcal{M}$. Each row $r \in$ table(h) is composed by the following components:*
  - *for each $v \in h$ with label$(v) = a$, a list $PL[a]$ of pointers to log tuples.*
  - *previous is a pointer to a table row (possibly null).*
  - *completed is a boolean flag.*
  - *model is the identifier of a model in $\mathcal{M}$.*
  - *$minTS$ is a timestamp.*
- *$\mathcal{I}$ is a set containing, for each model $M \in \mathcal{M}$, a list instances(M) of pointers to table rows.*

The components of the rows of a table *table*(h) represent the following information:

- each pointer list $PL[a]$ contains pointers to log tuples l with $l.type = a$ or $l.type = a'$ and $(a', a) \in \Delta^+$ that are a part of the log segment which satisfies h.
- *previous* points to a row of *table*(h') with $h' \in in_M(h)$ – this row represents a segment to which the log tuples pointed by the lists in r can be linked in order to extend a partial instance.
- *completed* is *true* iff r represents a segment that completely satisfies h.
- *model* is the identifier of a model containing h.
- $minTS$ is the minimum $l.timestamp$ where l is a log tuple pointed by a list $PL[x]$ for each x such that $\exists v \in h$ with $label(v) = x$

The pointers in *instances*(M) point to table rows that represent log segments corresponding to terminal hyperedges in M which have been completed.

Example 3.19. Figure 3.6 shows the status of the AM-Index after indexing the log of Example 3.8 up to tuple ℓ_{10}. At this stage, the index contains the following information.

- The row in *table*(h_1) represents the fact that hyperedge h_1 of model M is completed by a segment that includes the log tuple ℓ_1 (that is instance of vertex v_1 of h_1). In addition, $minTS = 0$ means that the segment starts at time 0.
- The row in *table*(h_2) represents the fact that hyperedge h_2 of model M is completed by a segment that includes log tuples ℓ_1 and ℓ_4. Moreover, the *previous* pointer points to the row in *table*(h_1) as the segment represented by the row in *table*(h_2) follows the former in the (partial) attack instance being represented.
- The row in *table*(h_3) represents the fact that hyperedge h_3 of model M is completed by a segment that includes log tuples ℓ_4 and ℓ_5. The *previous* pointer points to the row in *table*(h_2) as the segment represented by the row in *table*(h_3) follows the former in the (partial) attack instance being represented.

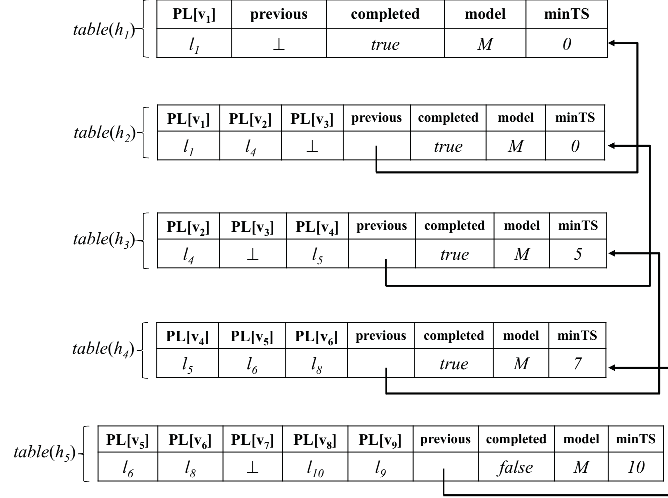


Fig. 3.6. Status of the AM-Index after indexing the log of Example 3.8 up to tuple ℓ_{10}

- The row in $table(h_4)$ represents the fact that hyperedge h_4 is completed by a segment that includes log tuples ℓ_5, ℓ_6 and ℓ_8. The *previous* pointer points to the row in $table(h_3)$ as the segment represented by the row in $table(h_4)$ follows the former in the (partial) attack instance being represented.
- The row in $table(h_5)$ represents the fact that hyperedge h_5 is partially completed by a segment that includes log tuples ℓ_6, ℓ_8, ℓ_9 and ℓ_{10}, but we still have to encounter a log tuple corresponding to vertex v_7.

Figure 3.7 shows the pseudo-code of the AM_Insert algorithm that indexes a new log tuple l_{new} with associated action $l_{new}.type$. In the algorithm, Lines 3-10 deal with the case where $l_{new}.type$ is in a start hyperedge. Specifically, the algorithm creates a new row r pointing to l_{new}, then adds this row to $table(h)$ (where h is the hyperedge containing $a = l_{new}.type$), and checks whether r represents a segment that completely satisfies h (by calling the CheckCompleted procedure). Lines 11-23 check whether l_{new} can be appended to the existing segments (i.e., every row $r \in table(h)$ with $r.completed = false$), by verifying cardinality and temporal constraints. If so, a new pointer to l_{new} is added to the pointer list $r.PL[a]$, and the new segment is passed to the CheckCompleted procedure. Otherwise, r is removed as it represents a non-extendable segment. Procedure CheckCompleted takes as input a table row r and an hyperedge h, and checks whether the segment of log tuples pointed by r is an instance and/or if it can be further extended. More specifically, Line 26 sets $r.completed = true$ because, according to the constraints expressed by h, the segment of log tuples pointed by r completely satisfies h. Line 27 stores a new instance by adding a pointer to r in $instances(M)$ if h is a terminal

```
Algorithm AM_Insert(l_new, I_M)
Input: New log tuple l_new, AM-Index I_M
Output: Updated AM-Index I_M
1  a ← l_new.type
   // —— Is a in a start hyperedge for some M ∈ 𝓜? ——
2  R ← ∅
3  for each M = ⟨𝓗, λ, τ, ε, S, T⟩ ∈ 𝓜
4    for each (h, v) s.t. h ∈ S, v ∈ h, and (label(v) = a or label(v) = a' and (a, a') ∈ Δ⁺)
5      r ← ⟨PL = ∅, ⊥, false, M, l_new.timestamp⟩
6      r.PL[a] ← {l_new↑}
7      add r to table(h) and to R
8      checkCompleted(h, r)
9    end for
10 end for
   // —— Can l_new be added to an existing segment? ——
11 for each M = ⟨𝓗, λ, τ, ε, S, T⟩ ∈ 𝓜, 𝓗 =< H, V >
12   for each (h, v) s.t. h ∈ H, v ∈ h, and (label(v) = a or label(v) = a' and (a, a') ∈ Δ⁺)
13     for each table row r ∈ table(h) \ R s.t. r.completed = false
14       if l_new.timestamp − r.minTS + 1 ≤ u' where τ(h) = (·, u')
15               and size(r.PL[a]) + 1 ≤ u where λ(v) = (·, ·, u)
16         add l_new↑ to r.PL[a]
17         checkCompleted(h, r)
18       else // —— The segment violates the constraints ——
19         remove r from table(h)
20       end if
21     end for
22   end for
23 end for
Procedure checkCompleted(h, r)
Input: Hyperedge h, index table row r
24 if l' ≤ l_new.timestamp − r.minTS + 1 where τ(h) = (l', ·)
25         and ∀w ∈ h, l_w ≤ size(r.PL[label(w)]) where λ(w) = (·, l_w, ·)
26   r.completed ← true
27   if h ∈ T then add r↑ to instances(M) // —— M is the model to which h belongs ——
     // —— Create the rows for outgoing hyperedges ——
28   for each h' s.t. h ∈ in_M(h')
29     r' ← ⟨PL = ∅, r↑, false, M, 0⟩
30     for each a_z s.t. ∃z ∈ h ∩ h' with label(z) = a_z
31       r'.PL[a_z] ← r.PL[a_z]
32     end for
33     r'.minTS ← minimum timestamp of the log tuples pointed by r'
34     add r' to table(h')
35   end for
36 end if
```

Fig. 3.7. AM_Insert algorithm.

hyperedge. In lines 28-35, for each outgoing hyperedge h' of h, a new row r' is created with $r'.previous = r$ and $r'.PL[a_z] = r.PL[a_z]$ for each $a_z \in h \cap h'$. In other words, if h has at least one outgoing hyperedge h', then h and h' have at least one vertex in common a_z, thus we must create table rows r in $table(h)$ and r' in $table(h')$, both pointing the same log tuple l with $l.type = a_z$.

Example 3.20. Figure 3.8 shows the evolution of the AM-Index when indexing tuples ℓ_{11} and ℓ_{12} of the log of Example 3.8. When indexing tuple ℓ_{11} (Figure 3.8(a)) the AM_Insert algorithm identifies the row in $table(h_5)$ as one that represents a segment that can be extended by ℓ_{11} (ℓ_{11} is an instance of vertex v_7). Thus, the algorithm replaces the $\perp$ pointer in $PL[v_7]$ with a pointer to ℓ_{11} (note that, for clarity of presentation, the figure shows a deletion of the row). Then, procedure CheckCompleted verifies that the segment is completed, so it marks the row as completed and creates a new row in $table(h_6)$ that contains, for each vertex in common between h_5 and h_6, a copy of the pointer list of the vertex. This new row gets completed after indexing ℓ_{12}, that is an instance of vertex v_{11} (Figure 3.8(b)). Moreover, since h_6 is a terminal hyperedge, a pointer to the row is added to $instances(M)$.

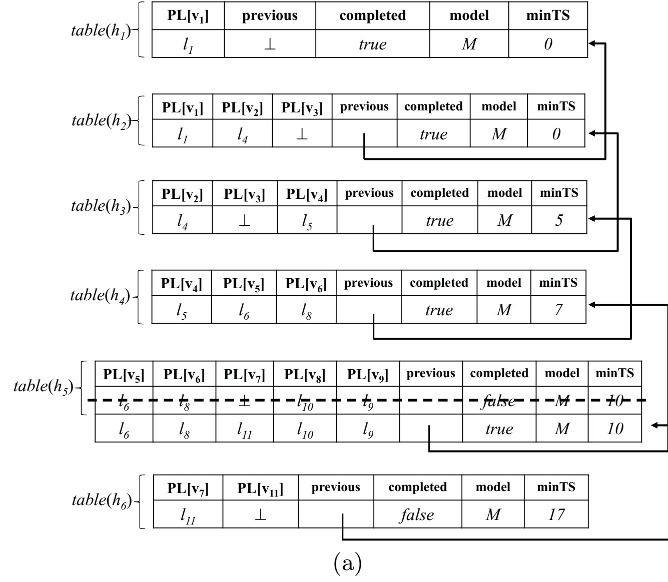
Figure 3.9 shows the pseudo-code of the AM_Retrieve algorithm that returns the instances of a set of attack models w.r.t. a log, represented as tuples of the form (M, l_s, l_t, π), where l_s and l_t are the first and last tuple of an instance of M over a complete path π. The algorithm takes as input an AM-Index $I_{\mathcal{M}} = \langle \mathcal{T}, \mathcal{I} \rangle$ and a set of models $\mathcal{M}' \subseteq \mathcal{M}$ such that for each $instances(M_i) \in \mathcal{I}$, $\bigcup_i M_i = \mathcal{M}'$. For each pointer $r^{\uparrow} \in instances(M)$, the algorithm assignes to l_t the most recent log tuple pointed by r, i.e., the last tuple of a log segment that completely satisfies a terminal hyperedge (Line 4). On Line 5, π is the set of hyperedges involved in the current instance. The algorithm follows the chain of $r.previous$ pointers backwards, from r to the row representing a start hyperedge (Lines 7-10). The last row r_{last} of the chain will be recognized due to its *previous* attribute set to $\perp$. This way, hyperedges represented by all rows of the chain are added to π. Finally, Line 11 assigns to l_s the less recent log tuple pointed by r_{last}, i.e., the first log tuple from which the current instance begins.

Finally, we point out that a simple yet effective concurrent pruning process can be performed during the maintenance of the AM-Index. In fact, when the AM_Insert algorithm eliminates a non-completed row from a table $table(h)$ because it violates the constraints, we can follow the chain of *previous* pointers starting from that row and eliminate any row $r \in table(h')$ if the following conditions hold:

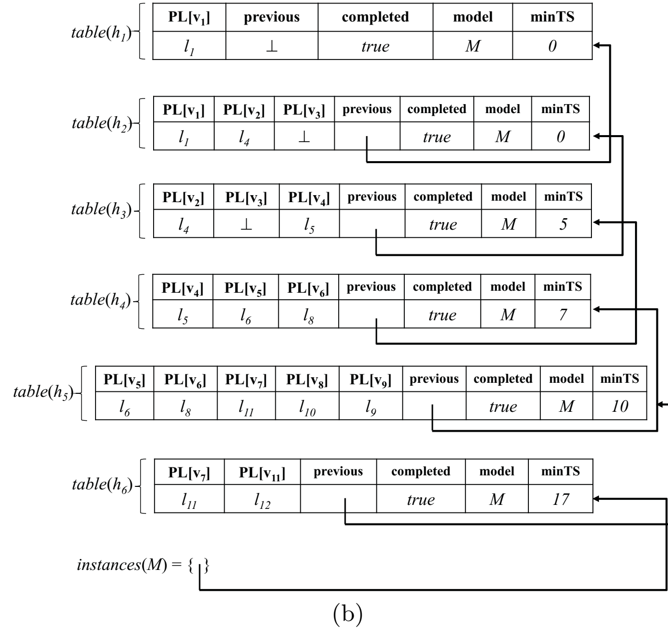
1. r is not pointed by any other row (this can be done efficiently by storing the number of pointers to each row);
2. r cannot be extended by a row corresponding to a segment for a hyperedge $h'' \neq h$ such that $h' \in in_M(h'')$ for some M.

3.7 Experimental Evaluation

In this section we report on the experimental results of our proposed algorithm when applied to a number of distinct attack models. We implemented the framework in Java. All the experiments were run on an Intel Core i7-3770K CPU clocked at 3.50GHz, running Windows 8, with 12GB RAM available.



(a)



(b)

Fig. 3.8. Evolution of the AM-Index when indexing tuples ℓ_{11} and ℓ_{12} of the log of Example 3.8

	Algorithm AM_Retrieve($I_{\mathcal{M}} =< \mathcal{T}, \mathcal{I} >, \mathcal{M}'$) **Input:** AM-Index $I_{\mathcal{M}}$, set $\mathcal{M}' \subseteq \mathcal{M}$ of attack models **Output:** Set I of tuples of the form (M, l_s, l_t, π) such that $M \in \mathcal{M}'$ and $l_s, \ldots, l_t \models_\pi M$
1	$I \leftarrow \emptyset$
2	**for each** $M \in \mathcal{M}'$
3	**for each** $r^\uparrow \in instances(M)$
4	$l_t \leftarrow$ most recent log tuple pointed by r
5	$\pi \leftarrow \{$hyperedge represented by $r\}$
6	$r_{curr} \leftarrow r$
7	**while** $r_{curr}.previous \neq \bot$
8	$r_{curr} \leftarrow r_{curr}.previous$
9	add the hyperedge represented by r_{curr} to π
10	**end while**
11	$l_s \leftarrow$ less recent log tuple pointed by r_{curr}
12	add (M, l_s, l_t, π) to I
13	**end for**
14	**end for**
15	**return** I

Fig. 3.9. AM_Retrieve algorithm.

3.7.1 Setting

We ran two rounds of experiments:

1. In the first round, we used the real-world attack model of Figure 3.1 along with other three real-world models.
2. In the second round, we used four models derived from those used in the first round by adding cardinality constraints, in order to analyze how the AM_Insert algorithm behaves when working with more constrained attack models.

For both rounds, we built synthetic logs consisting of 100K tuples. Each log was built by combining a set of sub-logs, each of which is a sequence of action symbols that can represent an instance of a given attack model. Specifically, a log combines several sub-logs $L_1, \ldots, L_n$ where each L_i is built by considering a complete path in a model. These sub-logs were built and combined under six different *log generation modes*, each corresponding to a possible real-world scenario:

1. each sub-log only contains action symbols in its corresponding model, and the sub-logs are concatenated;
2. same as mode 1, except that some action symbols are *substituted* with "noise", i.e. with symbols not present in the corrisponding model, with a certain frequency;
3. same as mode 1, except that noise is *inserted* in the sequence;

4. same as mode 1, except that a certain percentage of each L_i partially overlaps with L_{i+1};
5. same as mode 2, but with partial overlap as in mode 4;
6. same as mode 3, but with partial overlap as in mode 4.

We performed 14 runs for each round of experiments. The log generation modes, noise frequency, and overlap percentage used are reported in Figure 3.10. Note that default values are log generation mode 6 (that is the most complex mode, which also captures the fact that noise more often appears *between* alert symbols of actual interest), noise frequency 3/10, and overlap percentage 40%.[4]

Run	Log gen. mode	Noise freq.	Overlap perc.
1	1	–	–
2	2	3/10	–
3	3	3/10	–
4	4	–	40%
5	5	3/10	40%
6	6	3/10	40%
7	6	1/10	40%
8	6	2/10	40%
9	6	3/10	40%
10	6	4/10	40%
11	6	5/10	40%
12	6	3/10	20%
13	6	3/10	30%
14	6	3/10	40%
15	6	3/10	50%
16	6	3/10	60%

Fig. 3.10. Parameter values used for each experimental run.

We also performed experiments with much larger logs (1M tuples) – the performance we obtained in terms of tuples processed per second was 25% worse in the average case.

3.7.2 Results

Figure 3.11 reports the number of log tuples processed per second in both rounds of experiments. In particular, Figure 3.11(top) shows the number of tuples processed when varying the log generation mode (runs 1–6), Figure 3.11(center) shows the variation with respect to noise frequency (runs 7–11), and Figure 3.11(bottom) the variation with respect to the percentage of overlap between consecutive instances in the log (runs 12-16).

[4] For simplicity of presentation, the run with all parameters set to default values is reported as three separate runs (6, 9, and 14) in Figure 3.10.

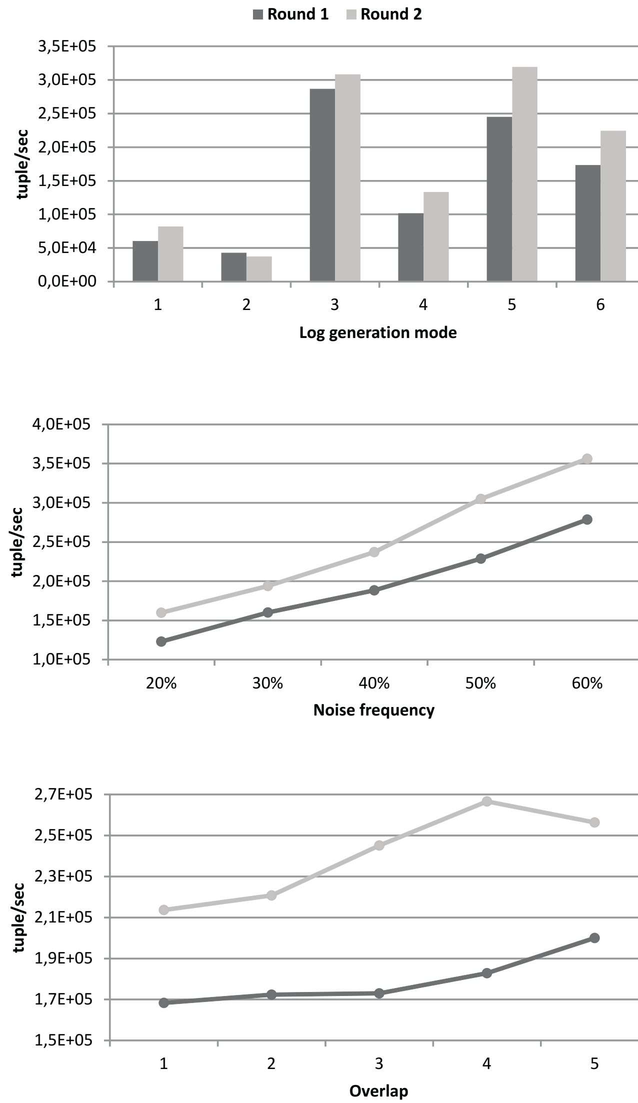


Fig. 3.11. Tuple rates in both rounds of experiments.

The results confirm our expectations and show very satisfying overall performances. In Figure 3.11(top) we can notice that, in the majority of cases, the presence of noise in the log or overlap between consecutive instances reduces the overall number of instances, thus improving performances. Generally, the number of tuples processed per second is very high – it is consistently higher than 42K, and over 151K on average. In both Figure 3.11(center) and Fig-

ure 3.11(bottom) the trend is almost linear in the frequency of noise and percentage of overlap – in these experiments, the average tuple rate is around 216K tuples/sec. When we look at the differences between the first and second round, i.e., when moving to more constrained attack models that allow for fewer instances, the results again appear satisfactory. – the performance gain is around 25% in the majority of cases.

We also measured the number of instances and the indexing time per tuple normalized by the number of instances. The results for all the runs are reported in Figure 3.12. As expected, the number of instances is much lower in the second round. Interestingly, the normalized indexing time shows relatively small variations with respect to the specific configuration used in the first round, and the framework appears to use slightly more resources per instance when using more constrained models.

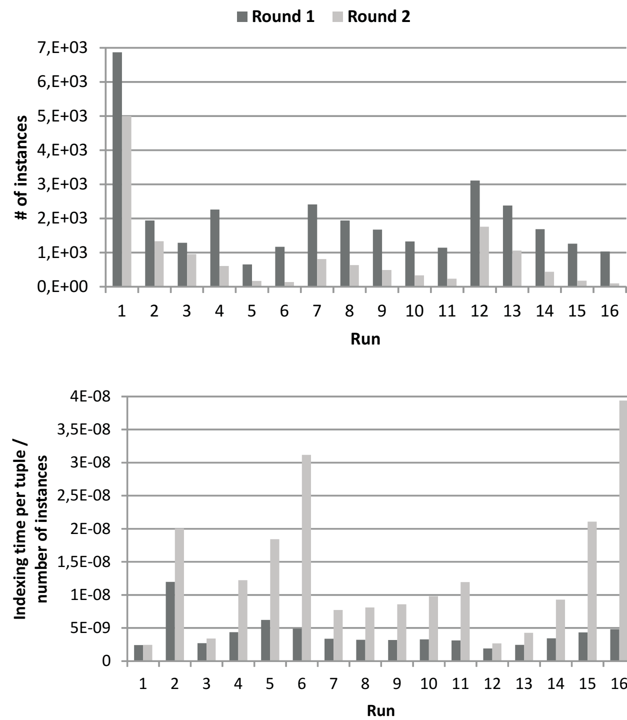


Fig. 3.12. Number of instances (top) and normalized indexing time per tuple (bottom) in both rounds of experiments.

Finally, Figure 3.13 reports the maximum size and the normalized maximum size of the AM-Index for all the runs. Interestingly, the overall size of the AM-Index appears relatively stable with respect to the configuration used.

Again, the framework appears to use more resources per instance when using more constrained models.

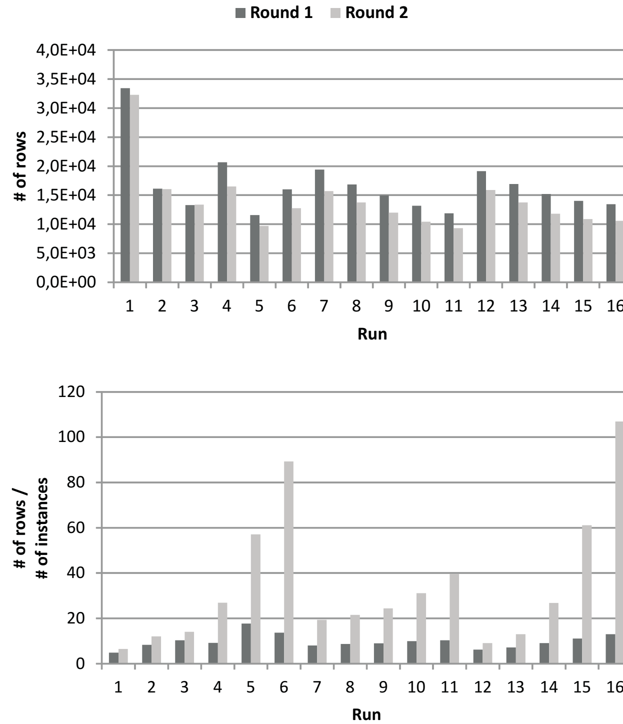


Fig. 3.13. Maximum size (top) and normalized maximum size (bottom) of the AM-Index in both rounds of experiments.

3.8 Related Work

3.8.1 Hypergraphs in security.

Hypergraphs [47] have been used to model networks in the security scenario. For instance, networks are modeled as hypergraphs in [13, 48] for network-based intrusion detection. In [49], hypergraphs are used for alert correlation, whereas [50] uses hypergraphs to model security dependencies in the context of risk analysis. In [51], a hypergraph-based model is presented for describing security properties, which allow focusing on the connections between entities, by generating intrusion scenarios in situations that transcend the physical

containment. In contrast to these works, we use hypergraphs to describe attacks instead of networks. A similar approach has been used in [52], where a datalog-based architecture is defined for a system that is able to reason on violations of the logging infrastructure. Instead of using an attack graph, they generalize the concept by representing the attack as a directed hypergraph, which allow to specify logical statement that describe the violation.

3.8.2 Workflow modeling.

Since an intrusion can be generally interpreted as a sequence of actions, many different kinds of models have been used in the past, such as, e.g., graph grammars [53] and Petri nets [54]. A number of works in the field of process modeling share with us the core idea of exploiting a hypergraph structure as generalization of a traditional (process) graph model. Unlike regular graph-structured process models, where an edge defines the sequential execution of adjacent vertices, an hyperedge is an arbitrary set of vertices which can be accomplished in any order. The applicability of hypergraphs to process modeling was studied, e.g., in [55, 56, 57, 58], where the authors extend a metagraph structure, originally proposed in [59] to represent transformation relations between two sets of objects, into a novel model, named *metagraph-based workflow*, tailored to model workflow executions. In contrast to traditional process modeling approaches, this approach proposes a view of workflow where actions are represented by edges that relate objects (i.e., vertices of the hyperedge) that are consumed and produced during action execution. Also inspired by hypergraph formalism, the approach in [60] uses hypergraphs to define execution semantics of *flexible process graphs* (FPG), a formal approach for modeling business processes in a flexible way. However, in a FPG, actions are represented by vertices rather than edges, and hyperedges define routing decisions by specifying executions of sets of actions that can be accomplished in an order that is actually determined when executing a process instance. Besides the specificity of the above models, the main difference with our proposed model is the focus on process modeling issues and their use for simulation/analyzing purposes. On the contrary, in our case hypergraphs are a basis for a completely new model to handle the problem of detecting intrusions.

3.8.3 Intrusion detection with attack graphs.

Our model fits well at the base of intrusion detection tasks that, to date, have often been based on direct attack graphs. In many works, such graphs are constructed by analyzing the interdependencies between vulnerabilities and security conditions that have been identified in the target network [16, 6, 61], or for correlating intrusion alerts [62, 63, 64]. One of the main advantages of using hypergraphs lies in the possibility of compactly representing a larger number of possible attack scenarios.

Finally, our idea to abstract an attack scenario using a hierarchy of action types is related to that proposed in [65], where associating a label to each attack action is proposed, in order to report the stage where the attacker is. However, differently from our purposes, they present a quantitative threat modeling method which quantifies security threats by calculating the total severity weights of attack paths that are considered to be relevant.

3.9 Conclusions

In this Chapter we proposed a hypergraph-based attack model for intrusion detection which provides a very high flexibility in capturing diverse security scenarios. We studied the problems of checking the consistency of attack models and detecting attack instances in sequences of logged actions. Moreover, we proposed an index structure and its associated maintenance and retrieval algorithms, whose efficiency has been proven by an extensive experimental evaluation.

Part II

Anomaly Detection

4

PADUA: Parallel Architecture to Detect Unexplained Activities

There are numerous applications (e.g., video surveillance, fraud detection, cybersecurity) in which we wish to identify unexplained sets of events. Most past work has been domain-dependent (e.g., in video surveillance or cybersecurity) and much of it focused on the valuable class of *statistical anomalies* in which statistically unusual events are considered. In contrast, assume that there is a set $\mathcal{A}$ of known activity models (both harmless and harmful) and a log L of time-stamped observations. We define a part $L' \subseteq L$ of the log to represent an *unexplained situation* when none of the known activity models can explain L' with a score exceeding a user-specified threshold. We represent activities via the notion of a *probabilistic penalty graph* (PPG) and show that a set of PPGs can be combined into one Super-PPG. We define an index structure for Super-PPGs. Given a compute cluster of $(K + 1)$ nodes (one of which is a master node), we show how to split a Super-PPG into K subgraphs that can be autonomously processed by K compute nodes. We provide algorithms for the individual compute nodes to ensure seamless handoffs that maximally leverage parallelism. PADUA is domain-independent and can be applied to many domains (perhaps with some specialization). We conducted detailed experiments with PADUA on two real-world datasets. First, we tested PADUA on the ITEA CANDELA video surveillance dataset. Second, we tested PADUA on network traffic data appropriate for cybersecurity applications. PADUA scales extremely well with the number of processors and significantly outperforms past work both in accuracy and time. Thus, PADUA represents the first parallel architecture and algorithms for identifying unexplained situations in observation data and—in addition to high accuracy—can scale well.

4.1 Introduction

Many organizations continuously monitor transactional data in order to identify irregularities. For instance, security officers at airports need to identify unexplained behavioral patterns (e.g., people who leave unattended packages)

in order to identify threats to public safety. Banks monitor transaction streams on their secure web sites to identify suspicious behaviors. Insurance companies look for unexplained patterns in claims data. Stock market regulators look for suspicious trading patterns that may artificially drive stock prices up or down [66]. In computer security, attack graphs [6] have been developed in order to identify known attack patterns by which hackers try to compromise systems. Insurance investigators have also developed patterns of activity to look for [67] . In all of these applications, experts have identified "known" patterns to look for. These known patterns include both harmless and harmful behavior—much work has focused on learning patterns of behavior [68, 69, 70, 71, 72, 73, 74, 75, 76, 77] so that statistically significant variations of these known patterns can be flagged.

"Bad guys" are constantly innovating and seeking new ways to carry out their crimes. In this chapter, we propose PADUA, the first parallel architecture for the detection of unexplained "situations" that we are aware of. A situation is any subset of a given log of observations. PADUA starts with some set $\mathcal{A}$ of activity models that are *known* in advance. $\mathcal{A}$ may consist of a combination of harmless and harmful activities—when a known harmful activity is detected in a log of observations, PADUA will automatically raise an alert that a security analyst can respond to (or a program specialized to address that harmful activity can be invoked). However, this chapter focuses on the problem of identifying situations that collectively cannot be satisfactorily explained by any of the activity models in $\mathcal{A}$. The contributions of this chapter are as follows.

1. Though the goal of PADUA is not to propose a new activity model, in Section 4.3 we propose *Probabilistic Penalty Graphs* (PPGs for short). PPGs extend stochastic automata [78] in order to handle "noise". Handling noise is critical in real-world applications as many observed events are probably irrelevant. For instance, in an airport, people exhibit a number of behaviors that were not thought of when models of known activities were developed. Similarly, activity patterns at a bank web site may exhibit a lot of noise. A situation is deemed unexplained if, intuitively, the situation cannot be explained by any known activity model in $\mathcal{A}$ with a score exceeding a user-defined threshold τ.
2. In Section 4.4, we propose a data structure that combines a set of PPGs into a single *Super-PPG*, together with algorithms to maintain this Super-PPG data structure and to seamlessly flag unexplained situations when they occur. Super-PPGs offer scalability on single machine implementations.
3. In Section 4.5, we show that given a set of $(K + 1)$ cluster nodes, we can split a Super-PPG into a set of K sub-PPGs, each of which can be executed on one of K compute nodes. We show 5 different approaches to splitting a Super-PPG for the purpose of detecting unexplained situations.

4. In Section 4.6, we provide parallel coordination algorithms for detecting unexplained situations using which each compute node can "hand off" computation to an appropriate other node when necessary.
5. We implemented all these data structures and algorithms on a parallel architecture with over 160 CPUs and conducted accuracy and timing experiments with two real-world datasets. The ITEA CANDELA dataset[1] contains a wide range of video surveillance data. The Naples Network Traffic dataset contains network traffic from the University of Naples. Experiments (reported in Section 4.7) show that: (*i*) PADUA scales extremely well with the number of CPUs and runs faster than past work for detecting unexplained situations, (*ii*) PADUA significantly improves the accuracy of past work [6]—the F-measure increases from 0.72 to 0.89.

We emphasize that an unexplained situation may not be a harmful one. For instance, if PADUA flags a situation as being unexplained, a security analyst may look at the situation—if it is harmless, it can be added to the set $\mathcal{A}$ of known activities as a harmless activity and if it is harmful, likewise, it can be added to the set $\mathcal{A}$ and flagged as harmful. By flagging unexplained situations we have the potential for a semi-automated method to grow the set of known activity patterns over time.

4.2 Related Work

4.2.1 A priori definitions.

Several researchers have studied how to search for specifically defined patterns of normal/abnormal activities using different models such as Hidden Markov Models [79, 80], coupled Hidden Markov Models [81, 82], Dynamic Bayesian Networks [83], Stochastic Automata [84], Bayesian networks and probabilistic finite state automata [85]. In contrast, this chapter starts with a set $\mathcal{A}$ of known activity models (for normal/abnormal activities) and finds sequences that are not sufficiently explained by any of the known models in $\mathcal{A}$.

4.2.2 Learning and then detecting abnormality.

Much work first learns a normal activity model and then detects abnormalities. In data mining, objects (e.g., people, transactions) may have an associated vector of properties. By clustering a set of objects, we can identify objects that either do not belong to any cluster or are "far away" (in the high-dimensional vector space) from any cluster. Here, belonging to a sufficiently big cluster is considered "normal", being far away from a cluster or being part of a tiny cluster is considered anomalous. Examples of such an approach are encompassed

[1] http://www.multitel.be/image/research-development/research-projects/candela/

in [86]. [68] proposes a semi-supervised approach to detect abnormal events that are rare, unexpected, and relevant. Detection of unseen or rarely occurring events are also considered in [69, 70, 71, 72, 73]. [87] defines an anomaly as an atypical behavior pattern that is not represented by sufficient samples in a training dataset and satisfies an abnormal pattern. [74] learns patterns of activities over time in an unsupervised way. [75] learn trajectory prototypes and detect anomalous behaviors when visual trajectories deviate from the learned representations of typical behaviors. [76] automatically learn high frequency events and declares them normal—events deviating from these rules are anomalies. [77] first analyzes and designs features from the data and then detects abnormal activities using the designed features. All these approaches first learn normal activity models and then detect abnormal/unusual events. These papers differ from this chapter as they consider rare events to be abnormal. In contrast, we may consider situations to be unexplained even if they are not rare—if existing models do not capture them with high probability, they are flagged as unusual. In addition, if a model exists for a rare situation, we would flag it as "explained", while many of these frameworks would not.

4.2.3 Similarity-based abnormality.

[88] proposes an unsupervised technique in which each event is compared with all other observed events to determine how many similar events exist. Unusual events are events for which there are no similar events. Similarly, [89] considers a scene in a video anomalous when the maximum similarity between the scene and all previously viewed scenes is below a threshold. In [90], frequently occurring patterns are normal and patterns that are dissimilar from most patterns are anomalous. An unsupervised approach, where an abnormal trajectory refers to something that has never (or rarely) been seen was proposed in [91]. In [92], unusual events are detected by computing the likelihood of a new observation w.r.t. the probability distribution of prior observations.

4.2.4 Cybersecurity.

Intrusion detection systems (IDSs) monitor network traffic for suspicious behavior and trigger alerts [8, 93, 94]. Alert correlation methods aggregate such alerts into multi-step attacks [22, 6, 95, 96]. Intrusion detection techniques can be broadly classified into *signature-based* [94] and *profile-based* (or *anomaly-based*) [8] methods. A signature is a set of conditions that characterize intrusion activities w.r.t. packet headers and payload content. Historically, signature-based methods have been used extensively to detect malicious activities. In profile-based methods, a known deviation from the norm is considered anomalous (e.g. HTTP traffic on a non-standard port). In contrast, in this chapter, we consider the case where we have a set $\mathcal{A}$ of known activities (both innocuous and dangerous)—and we are looking for observation sequences that cannot be explained by either (if they were, they would constitute patterns

that were known a priori). These need to be flagged as they might represent "zero day" attacks. Correlation techniques try to reconstruct attacks from isolated alerts. The main role of correlation is to provide a higher level view of the actual attacks [95, 96, 97, 98, 99]. Both IDSs and correlation techniques rely on models encoding a priori knowledge of either normal or malicious behavior, and cannot appropriately deal with events that are not explained by the underlying models.

4.3 Probabilistic Penalty Graphs

We assume the existence of a finite set $\mathcal{A}$ of *action symbols*. A *log tuple* is a $(k+1)$-tuple $l = (s, att_1, \ldots, att_k)$ where $s \in \mathcal{A}$, and $att_1, ..., att_k$ are attributes (e.g., source, actor, time-stamp etc.). A *log* is a finite sequence of log tuples. We use $l.action$ to refer to action symbol s of tuple l. Intuitively, a log tuple corresponds to an observation of $l.action$ along with the associated attributes of the observation $att_1, \ldots, att_k$. By convention, if action a_2 occurs after a_1 in a log, then the action a_2 occurred temporally after a_1.

4.3.1 Definition of PPGs

In this chapter, we model activities using *probabilistic penalty graphs* which extend the stochastic activity definition of [78] with a penalty component which allows us to handle noise.

Definition 4.1 (Probabilistic Penalty Graph (PPG)). *A* probabilistic penalty graph *(PPG for short) is a labeled graph $A = (V, E, \delta, \rho)$ where:*

- $V \subseteq \mathcal{A}$ *is the set of nodes;*
- $E \subseteq V \times V$ *is the set of edges (with no self-loops);*
- *the set of start nodes (i.e., nodes with in-degree zero), denoted* $start(A)$*, is non-empty;*
- *the set of end nodes (i.e., nodes with out-degree zero), denoted* $end(A)$*, is non-empty;*
- $\delta : E \to (0, 1)$ *is a function that associates a probability distribution with the outgoing edges of each node, i.e.,* $\forall v \in V$, $\sum_{(v,v') \in E} \delta(v, v') = 1$*;*
- $\rho : E \to (0, 1)$ *is a function that associates a noise degradation value with each edge.*

The last component (noise degradation function) is new and extends the stochastic automata of [78]. To understand the intuition behind it, consider an edge $e = (a_1, a_2)$ in some PPG labeled with probability $\delta(e)$ and noise degradation $\rho(e)$. This edge can be read as: if a_1 occurs in a log and a_2 occurs later and there are z actions $b_1, \ldots, b_z$ in the log *strictly* between a_1 and a_2, then the score associated with the subsequence $a_1, b_1, \ldots, b_z, a_2$ is $\delta(e) \cdot \rho(e)^z$. As the degradation $\rho(e) \in [0, 1]$, the larger z is, the lower

the subsequence score (because $\rho(e)^z$ decreases as z increases). Thus, the subsequence "pays a penalty" as the amount of noise in it increases. For example, consider an edge $e = (a_1, a_2)$ with $\delta(e) = 0.7$ and $\rho(e) = 0.2$. Suppose our log contains the sequence $\langle a_1, b_1, b_2, a_2 \rangle$. The score of a transition from a_1 to a_2 is $0.7 * (0.2)^2 = 0.028$ because there are two "noisy" events (b_1, b_2) in the middle. In contrast, with the same δ and ρ, consider the log $\langle a_1, b_2, b_2, b_3, a_2 \rangle$ where an extra noisy observation b_3 occurs between a_1, a_2. The score of a transition from a_1 to a_2 is $0.7 * (0.2)^3 = 0.0056$, an even smaller number.

Example 4.2. The PPG in Fig. 4.1 shows an e-commerce network intrusion scenario from [6]. Here PostFirewallAccess is the only start node and CentralDBServerAccess is the only end node. Each edge e is labeled with $(\delta(e), \rho(e))$, i.e., the probability and noise degradation values for the edge. For example, the outgoing edges of node PostFirewallAccess show that there is a 90% probability the next action is MobileAppServerAccess and a 10% probability it is CentralDBServerAccess. Furthermore, the former edge has a degradation value 0.2 and the latter has a degradation value 0.4.

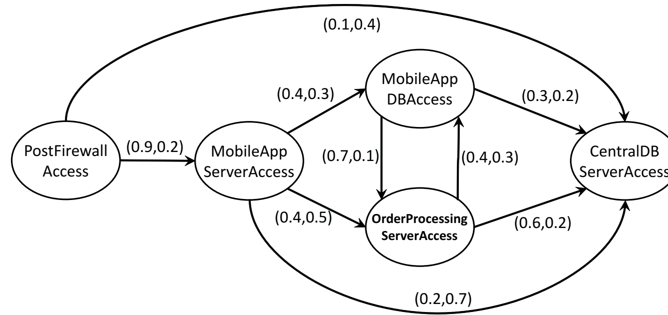


Fig. 4.1. Probabilistic Penalty Graph.

4.3.2 Unexplained Situations

In order to define unexplained situations, we first define PPG occurrences in a log.

Definition 4.3 (PPG Occurrence). *Let $A = (V, E, \delta, \rho)$ be a PPG and L a log. An* occurrence *of A in L is a pair (L^*, I^*) where*

1. *$L^* = l_1, \ldots, l_n$ is a contiguous subsequence of L;*
2. *$I^* = i_1, \ldots, i_m$ is a sequence of indices of L, with $1 \leq i_j \leq n$, $i_1 = 1$, $i_m = n$, and $i_j < i_{j+1}$;*
3. *$\forall j \in [1, m-1]$, $(l_{i_j}.action, l_{i_{j+1}}.action) \in E$;*
4. *$l_1.action \in start(A)$;*

5. $l_n.action \in end(A)$.

Thus, an occurrence of a PPG A consists of a *contiguous* subsequence L^* of L and a set I^* of indexes specifying the tuples in L^* whose associated actions are a path from a start to an end node in A—the remaining tuples of L^* constitute "noise". Fig. 4.2 illustrates the definition pictorially—each "box" represents a log tuple. In this example, $L^* = \langle l_1, l_2, \ldots, l_9\rangle$ consists of the entire sequence shown, while $I^* = \langle 1, 5, 6, 9\rangle$ refers to the shaded boxes. For this to be a valid occurrence of a PPG A, we need to make sure that A contains an edge from $l_1.action$ to $l_5.action$, from $l_5.action$ to $l_6.action$, and from $l_6.action$ to $l_9.action$, and moreover ensure that $l_1.action$ is a valid start state node and $l_9.action$ is a valid end node for A.

Fig. 4.2. An example of PPG occurrence.

Of course, some occurrences are "good" while others may not be as good. This "goodness" is captured via the score of an occurrence which is defined below. The *score* of (L^*, I^*) is

$$score(L^*, I^*) = \Pi_{j\in[1,m-1]}\delta(l_{i_j}.action, l_{i_{j+1}}.action)\cdot\rho(l_{i_j}.action, l_{i_{j+1}}.action)^z$$

with $z = i_{j+1} - i_j - 1$. Thus, the score of (L^*, I^*) is computed by taking into account

1. the probability on the edges belonging to the path $l_{i_1}.action, \ldots, l_{i_m}.action$ specified by I^* (i.e., $\Pi_{j\in[1,m-1]}\delta(l_{i_j}.action, l_{i_{j+1}}.action)$), and
2. the amount of noise in L^*.

If there are many tuples in L^* which are not part of a path from a start to an end node in A, then the score of (L^*, I^*) decreases. This is because when z (the amount of noise) increases, multiplying $\delta(l_{i_j}.action, l_{i_{j+1}}.action)$ by $\rho(l_{i_j}.action,\ l_{i_{j+1}}.action)^z$ yields a smaller score, because $\rho(-,-) \in [0, 1]$. We illustrate this below.

Example 4.4. Consider a log whose associated sequence of action symbols is ⟨PostFirewallAccess, x, MobileAppServerAccess, OrderProcServerAccess, x, x, CentralDBServerAccess, x⟩, where x $\notin V$ and V is the set of vertices of the PPG in Fig. 4.1. Then, (L^*, I^*), where L^* = ⟨PostFirewallAccess, x, MobileAppServerAccess, OrderProcServerAccess, x, x, CentralDBServerAccess⟩ and $I^* = 1, 3, 4, 7$, is an occurrence of the PPG in Fig. 4.1 and its score is $0.9 \cdot 0.2^1 \cdot 0.4 \cdot 0.5^0 \cdot 0.6 \cdot 0.2^2$.

Now we come to the critical definition of an unexplained situation.

Definition 4.5 (Unexplained Situation). *Let $A = (V, E, \delta, \rho)$ be a PPG and L a log. An* unexplained situation *for A is a pair (L_u, I_u) where*

1. *Conditions 1–4 in Definition 4.3 hold;*
2. *$l_n.action \in V - end(A)$;*
3. *there is no occurrence (L^*, I^*) of A such that L_u is a prefix of L^* and I_u is a prefix of I^*;*
4. *there is no pair $(L'_u, I'_u) \neq (L_u, I_u)$ such that L_u is a prefix of L'_u, I_u is a prefix of I'_u, and (L'_u, I'_u) satisfies all conditions above.*

Thus, an unexplained situation for A consists of a *contiguous* subsequence L_u of L and a set I_u of indexes specifying the tuples in L_u whose associated actions are a path ending in a non-end node in A. The third condition requires that an unexplained situation cannot be extended so as to get an occurrence of A. The fourth condition ensures that unexplained situations are as long as possible.

The score $score(L_u, I_u)$ of an unexplained situation (L_u, I_u) is given by:

$$\Pi_{j\in[1,m-1]}(1 - \delta(l_{i_j}.action, l_{i_{j+1}}.action)) \cdot (1 - \rho(l_{i_j}.action, l_{i_{j+1}}.action))^z$$

with $z = i_{j+1} - i_j - 1$. The score of (L_u, I_u) takes into account the probabilities of the edges along the path specified by I_u and the noise degradation values for the tuples in L_u which are not referenced by I_u; however, edge probabilities and degradation values are complemented. We illustrate this in the following examples.

Example 4.6. Consider the PPG in Fig. 4.1. Let δ be

- δ(MobileAppServerAccess, MobileAppDBAccess) $= 0.1$
- δ(MobileAppServerAccess, OrderProcServerAccess) $= 0.7$.

Now consider a log containing two subsequences of tuples

- $L_1 = \langle$MobileAppServerAccess, x, x, x, MobileAppDBAccess$\rangle$
- $L_2 = \langle$MobileAppServerAccess, x, x, x, OrderProcServerAccess$\rangle$

In this case, L_1 may contribute to the score of an occurrence of the PPG with a probability equal to 0.1, and L_2 with a probability of 0.7. This means that "moving" from MobileAppServerAccess to MobileAppDBAccess is less likely than moving from MobileAppServerAccess to OrderProcServerAccess in the activity described by the PPG. Hence, when computing the score of unexplained situations containing L_1 or L_2, the contributions are complemented: moving from MobileAppServerAccess to MobileAppDBAccess is considered "more unexplained" than moving from MobileAppServerAccess to OrderProcServerAccess, and their contributions to the score become 0.9 and 0.3, respectively.

Example 4.7. Consider the log ⟨PostFirewallAccess, x, MobileAppServerAccess, MobileAppDBAccess, x, x⟩. Then, (L_u, I_u), where $L_u = \langle$PostFirewallAccess, x, MobileAppServerAccess, MobileAppDBAccess$\rangle$ and $I_u = 1, 3, 4$ is an unexplained situation for the PPG in Fig. 4.1 with score $0.1 \cdot 0.8^1 \cdot 0.6 \cdot 0.7^0$.

We now define the concept of an unexplained situation w.r.t. a PPG when we are given a threshold τ.

Definition 4.8. *Let A be a PPG, L a log, and $\tau \in [0, 1]$. A* τ-unexplained situation *for A is an unexplained situation (L_u, I_u) for A with score$(L_u, I_u) \geq \tau$.*

The definition of a τ-unexplained situation above is given for a *single* PPG. Given a set $\mathcal{A}$ of PPGs, we would like to find a set of τ-unexplained situations *w.r.t. the whole set* $\mathcal{A}$. We define the τ-unexplained situations for a set of PPGs as follows.

Definition 4.9. *Let $\mathcal{A}$ be a set of PPGs, L a log, and $\tau \in [0, 1]$. A* τ-unexplained situation *for $\mathcal{A}$ is a maximal contiguous subsequence L_u of L such that for every A in $\mathcal{A}$, there is a τ-unexplained situation (L'_u, I'_u) for A s.t. L_u is a subsequence of L'_u.*

Thus, given a set $\mathcal{A}$ of PPGs, a τ-unexplained situation is a maximal contiguous subsequence L_u of the log which is contained in a τ-unexplained situation of every PPG in $\mathcal{A}$ (i.e., L_u is unexplained w.r.t. all PPGs in $\mathcal{A}$). Before computing the set of τ-unexplained situations, we show that our theory has several elegant properties.

As threshold τ is used to select only those unexplained situations for which we have a confidence above τ, higher values of τ are stricter conditions for a situation to be unexplained. The following proposition states that our framework satisfies the natural property that unexplained situations become wider by decreasing the threshold (moreover, new unexplained situations might be found).

Proposition 4.10. *Consider a log L, a set $\mathcal{A}$ of PPGs, and two thresholds $\tau_1, \tau_2 \in [0, 1]$. Let U_1 (resp. U_2) be the set of τ_1- (resp. τ_2-) unexplained situations for $\mathcal{A}$. If $\tau_1 \geq \tau_2$, then for every L^1_u in U_1 there exists L^2_u in U_2 s.t. L^1_u is a contiguous subsequence of L^2_u.*

Proof. Let L^1_u be a τ_1-unexplained situations for $\mathcal{A}$, i.e., $L^1_u \in U_1$. Definition 4.9 implies that for every $A \in \mathcal{A}$ there is a τ_1-unexplained situation (L'_u, I'_u) for A s.t. L^1_u is a subsequence of L'_u. It is easy to check that $\tau_1 \geq \tau_2$ implies that (L'_u, I'_u) is a τ_2-unexplained situation for A. This means that for every $A \in \mathcal{A}$ there is a τ_2-unexplained situation (L'_u, I'_u) for A s.t. L^1_u is a subsequence of L'_u. The following two cases may occur. *(i)* If L^1_u is maximal, then the claim trivially holds (we are in the case where $L^2_u = L^1_u$). *(ii)* If L^1_u is not maximal, then there exists $L^2_u \neq L^1_u$ s.t. L^1_u is a proper contiguous subsequence of L^2_u, and for every $A \in \mathcal{A}$ there is a τ_2-unexplained situation (L'_u, I'_u) for A s.t. L^2_u is a subsequence of L'_u.

Given two PPGs $A_1 = (V_1, E_1, \delta_1, \rho_1)$ and $A_2 = (V_2, E_2, \delta_2, \rho_2)$, we write $A_1 \sqsubseteq A_2$ iff *(i)* $V_1 = V_2$, *(ii)* $E_1 = E_2$, and *(iii)* $\delta_1(e) \leq \delta_2(e)$ and $\rho_1(e) \leq \rho_2(e)$ for every $e \in E_1$ (or, equivalently, $e \in E_2$). Given two sets $\mathcal{A}_1$ and $\mathcal{A}_2$ of PPGs we write $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ iff for every $A_1 \in \mathcal{A}_1$ there exists $A_2 \in \mathcal{A}_2$ s.t. $A_1 \sqsubseteq A_2$.

Intuitively, $A_1 \sqsubseteq A_2$ means that A_1 and A_2 are topologically the same, but A_2 has possibly higher edge probabilities or penalties. Notice that higher edge probabilities/penalties for a PPG lower the confidence we have in τ-unexplained situations and thus, we would expect a smaller portion of the log to be unexplained. Indeed, as stated by the following proposition, this is correctly captured by our theory.

Proposition 4.11. *Consider a log L, a threshold $\tau \in [0, 1]$, and two sets of PPGs $\mathcal{A}_1$, $\mathcal{A}_2$. Let U_1 (resp. U_2) be the set of τ-unexplained situations for $\mathcal{A}_1$ (resp. $\mathcal{A}_2$). If $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$, then for every L_u^1 in U_1 there exists L_u^2 in U_2 s.t. L_u^1 is a contiguous subsequence of L_u^2.*

Proof. Recall that, by definition of $\sqsubseteq$, if $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$, then for every $A_2 \in \mathcal{A}_2$ there exists $A_1 \in \mathcal{A}_1$ s.t. $A_2 \sqsubseteq A_1$. Notice that if (L_u, I_u) is a τ-unexplained situation for A_1, then (L_u, I_u) is a τ-unexplained situation also for A_2. This is because (L_u, I_u) is clearly an unexplained situation for both A_1 and A_2 (because they are topologically the same and the definition of unexplained situation involves only the topology of the PPG and the log—see Definition 4.5) and the score of (L_u, I_u) computed w.r.t. A_1 is less than or equal to the score of (L_u, I_u) computed w.r.t. A_2.

Suppose $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$ and let L_u^1 be a τ-unexplained situations for $\mathcal{A}_1$. Definition 4.9 implies that for every $A_1 \in \mathcal{A}_1$ there is a τ-unexplained situation (L'_u, I'_u) for A_1 s.t. L_u^1 is a subsequence of L'_u. Suppose by contradiction that there does not exist a τ-unexplained situation L_u^2 for $\mathcal{A}_2$ s.t. L_u^1 is a contiguous subsequence of L_u^2. In other words, neither L_u^1 nor any contiguous subsequence of the log containing L_u^1 is a τ-unexplained situations for $\mathcal{A}_2$. Then, by Definition 4.9, there is at least one PPG A_2 in $\mathcal{A}_2$ s.t. there does not exist a τ-unexplained situation (L'_u, I'_u) for A_2 s.t. L_u^1 is a subsequence of L'_u. The definition of $\sqsubseteq$ implies that there exists $A_1 \in \mathcal{A}_1$ s.t. $A_2 \sqsubseteq A_1$. Since L_u^1 is a τ-unexplained situations for $\mathcal{A}_1$, then there must be a τ-unexplained situation (L'_u, I'_u) for A_1 s.t. L_u^1 is a subsequence of L'_u. As shown at the beginning of the proof, this means that (L'_u, I'_u) is a τ-unexplained sequence for A_2, and, furthermore, L_u^1 is a subsequence of L'_u, which is a contradiction.

As we wish to find situations that are not sufficiently explained by a set $\mathcal{A}$ of PPGs, another natural property is that a smaller portion of the log becomes unexplained by adding PPGs to $\mathcal{A}$. The following corollary says that this property is satisfied by our theory.

Corollary 4.12. *Consider a log L, a threshold $\tau \in [0, 1]$, and two sets of PPGs $\mathcal{A}_1$, $\mathcal{A}_2$. Let U_1 (resp. U_2) be the set of τ-unexplained situations for $\mathcal{A}_1$ (resp. $\mathcal{A}_2$). If $\mathcal{A}_2 \subseteq \mathcal{A}_1$, then for every L_u^1 in U_1 there exists L_u^2 in U_2 s.t. L_u^1 is a contiguous subsequence of L_u^2.*

Proof. It is straightforward to check that $\mathcal{A}_2 \subseteq \mathcal{A}_1$ implies $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$ (see the definition of $\sqsubseteq$). Thus, the claim follows from Proposition 4.11.

4.3.3 Deriving Noise Degradation Values from a Training Set

We can easily derive noise degradation values from a training set of data as follows. Suppose we have a PPG A and $e = (a_1, a_2)$ is an edge in this PPG. Suppose we have a log L. Let $occ(L, e)$ denote the set of all contiguous sequences in L that start with a_1 and end with a_2. Suppose these are presented to a user for training purposes and the user marks some of these as valid transitions from a_1 to a_2 and marks the others as invalid. Let $valid(L, e)$ be the subset of $occ(L, e)$ marked valid and $invalid(L, e) = occ(L, e) \setminus valid(L, e)$. For any sequence $s = a_1, b_1, \ldots, b_k, a_2$, let $noise(s, e) = k$. Moreover, for each integer i, let $f(i)$ be the percentage of sequences in $occ(L, e)$ with i units of noise between a_1 and a_2 that are marked valid, i.e.,

$$f(i) = \frac{|\{s \mid s \in valid(L,e) \wedge noise(s,e) = i\}|}{|\{s \mid s \in occ(L,e) \wedge noise(s,e) = i\}|}$$

We now plot a graph with i on the x-axis and $f(i)$ on the y-axis and look for a value $\rho(e)$ such that the function $g(i) = \delta(e) * \rho(e)^i$ best approximates the function f, i.e. such that the mean square error $\Sigma_i (g(i) - f(i))^2$ is minimized. This can be done by a standard curve fitting procedure [100]. [2]

As an example of this procedure, suppose we consider an edge $e = (a_1, a_2)$ and that the training set has a maximum of 3 noisy observations between a_1, a_2. Suppose the table below shows the set $occ(L, e)$ along with the valid/invalid annotation.

Sequence s	*noise(s)*	*Annotation*
a_1, b_1, a_2	1	valid
a_1, b_3, a_2	1	valid
a_1, b_1, b_2, a_2	2	valid
a_1, b_1, b_3, a_2	2	valid
a_1, b_1, b_4, a_2	2	invalid
a_1, b_1, b_2, b_3, a_2	3	valid
a_1, b_2, b_3, b_5, a_2	3	invalid
a_1, b_2, b_6, b_2, a_2	3	invalid

According to this training set, $f(1) = 1, f(2) = 0.67, f(3) = 0.33$. Suppose the transition probability $\delta(e)$ is 0.5. Then we are looking for a function $g(i) = \delta(e) * \rho(e)^i$ such that $\Sigma_{i=1}^{3} (g(i) - f(i))^2$ is minimized. Let $\rho(e) = u$. Then we want to minimize $(0.5 * u - 1)^2 + (0.5 * u^2 - 0.67)^2 + (0.5 * u^3 - 0.33)^2$. We can minimize this expression, subject to the requirement that it is non-zero.

[2] Note that this is just one simple way of learning penalties. The goal here is not to study machine learning algorithms to learn such graphs, just to show that reasonably simple ways to learn these penalties exist.

This is a straightforward polynomial (cubic) constraint solving problem that can be solved using any non-linear constraint solver.

4.4 The PPG-Index: Fast Computation of Unexplained Situations on a Single CPU

In this section, we define a PPG-Index to quickly compute the set of all τ-unexplained situations within a log. We first show how to merge all PPGs together into a Super-PPG. Then we develop a PPG-Index structure that is automatically updated when new observations come into the log. The PPG-Index is fully geared towards finding the τ-unexplained situations within a log as the log is changing. This section focuses on implementing these operations on a single CPU while subsequent sections define the parallel algorithms within PADUA.

4.4.1 Super-PPGs

We first define a *Super-PPG*, which is a compact representation of a set of PPGs. Note that a Super-PPG is not a PPG, but a slightly different structure which encapsulates all the information within the given set of PPGs.

Definition 4.13 (Super-PPG). *Let $\mathcal{A} = \{A_1, \ldots, A_k\}$ be a set of PPGs, where $\forall i \in [1,k]$, $A_i = (V_i, E_i, \delta_i, \rho_i)$. A* Super-PPG *is a 4-tuple $G(\mathcal{A}) = (V_G, E_G, \delta_G, \rho_G)$ where*

- $V_G = \cup_{i\in[1,k]} V_i$ *and* $E_G = \cup_{i\in[1,k]} E_i$*;*
- $\delta_G : V_G \times V_G \times \mathcal{A} \to [0,1]$ *is the function s.t.* $\delta_G(v, v', A_i) = \delta_i(v, v')$ *if* $(v, v') \in E_i$*, 0 otherwise.*
- $\rho_G : V_G \times V_G \times \mathcal{A} \to [0,1]$ *is the function s.t.* $\rho_G(v, v', A_i) = \rho_i(v, v')$ *if* $(v, v') \in E_i$*, 0 otherwise.*

Basically, the Super-PPG associated with $\mathcal{A}$ has the same vertices as in the graphs in $\mathcal{A}$. The “global” probability function δ_G returns the probability of an edge in this graph w.r.t. a specific activity and the “global” ρ_G noise degradation function does the same.

Example 4.14. Let $\mathcal{A} = \{A_1, A_2\}$ where A_1 is the PPG in Fig. 4.1 and A_2 is the PPG in Fig. 4.4. Fig. 4.3 shows $G(\mathcal{A})$, where each edge (v_1, v_2) has labels of the form $A : (\delta_G(v_1, v_2, A), \rho_G(v_1, v_2, A))$.

The definition of a Super-PPG yields an immediate way of quickly constructing a Super-PPG.

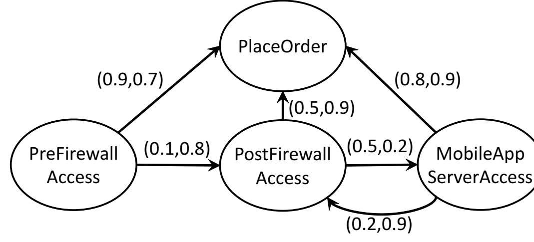


Fig. 4.3. PPG A_2.

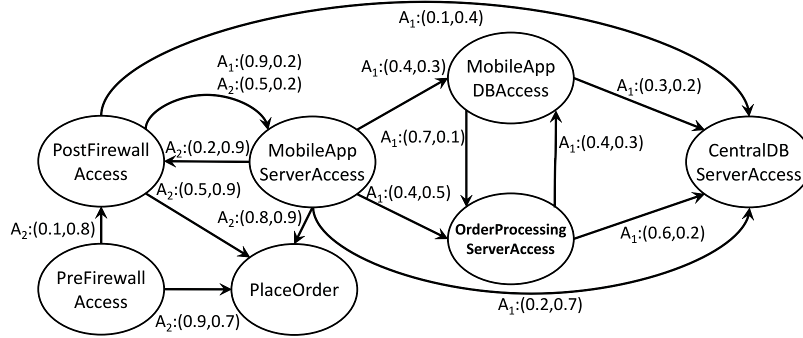


Fig. 4.4. Super-PPG.

4.4.2 The PPG-Index

The *PPG-Index* uses the Super-PPG to efficiently keep track of all unexplained situations found in a log whose score is above a threshold τ. In the following, we denote the set of references (pointers) to the elements in a set S as *ref*(S).

Definition 4.15 (PPG-Index). *Let $\mathcal{A}$ be a set of PPGs, L a log, and $G(\mathcal{A}) = (V_G, E_G, \delta_G, \rho_G)$. A* PPG-Index *is a 4-tuple $I_G = (G(\mathcal{A})$, $tables_G$, $count_G$, $completed_G)$, where:*

- *For each $v \in V_G$, $tables_G(v)$ is a set of tuples of the form (current, A, score, previous, closed, count), where current $\in$ ref(L), $A \in \mathcal{A}$, score $\in [0,1]$, previous $\in$ ref($\mathcal{P}$) with $\mathcal{P} = \bigcup_{v \in V_G} tables_G(v)$, closed is a boolean value, and count $\in \mathbb{N}$;*
- *$count_G \in \mathbb{N}$;*
- *$completed_G : \mathcal{A} \to 2^{ref(\mathcal{P})}$ is a function that associates each PPG with a set of references to the tuples in $tables_G$.*

For each action symbol $v \in V_G$, the index contains a table $tables_G(v)$. In the table, each tuple $t = (current, A, score, previous, closed, count)$ represents the fact that an unexplained situation for PPG A contains the log tuple pointed by *current*, and *current.action* $= v$. In particular:

- the score of the sequence up to the *current* tuple is equal to the value of *score*;
- *previous* points to the index tuple that precedes t in the sequence;
- *closed* indicates if the situation cannot be extended with a score above the threshold;
- *count* is the number of log tuples indexed before *current* (including *current* itself);
- $count_G$ is the global counter of indexed log tuples;
- $completed_G$ associates a PPG with those tuples in $tables_G$ that represent the last action of an unexplained situation.

When the log is empty, the PPG-Index is empty (with empty $tables_G$ and $completed_G$, and with $count_G = 0$)—we use I_G^0 to denote this "empty" PPG-Index.

Example 4.16. Let $\mathcal{A} = \{A_1, A_2\}$ be the set of PPGs of Example 4.14 and consider a log whose associated sequence of action symbols is ⟨PreFirewallAccess, x, PostFirewallAccess⟩. The corresponding index tables are shown in Fig. 4.5 (dashed box). The index contains an index tuple in $tables_G$(PreFirewallAccess) (denoted t_{eb} in the following) and two tuples in $tables_G$(PostFirewallAccess) (t_{ic}, t'_{ic} in the following). The presence of t_{eb} indicates that PreFirewallAccess is a start node for A_2: it has no *previous* index tuple and its *score* is 1. Moreover, its *count* is 1 because it is the first action in the log. Likewise, t_{ic} means that PostFirewallAccess is a start node for A_1. Finally, t'_{ic} means that PostFirewallAccess can also follow PreFirewallAccess in an unexplained situation for A_2. Thus, its *previous* index tuple is t_{eb} and its *score* is

$$t_{eb}.score \cdot (1 - \delta_G(\mathsf{PreFirewallAccess}, \mathsf{PostFirewallAccess}, A_2))$$
$$\cdot (1 - \rho_G(\mathsf{PreFirewallAccess}, \mathsf{PostFirewallAccess}, A_2))$$

where the penalty component derives from the presence of x between PreFirewallAccess and PostFirewallAccess. Note that at this point we have $count_G = 3$, $completed_G(A_1) = \{t_{ic}\}$, and $completed_G(A_2) = \{t'_{ic}\}$.

Fig. 4.6 shows the pseudo-code of the PPG_Insert algorithm that indexes a new log tuple l_{new} with associated action symbol $l_{new}.action = a$. In the algorithm, Lines 3–6 deal with the case where a is a start node for some PPG, by creating a new sequence. Lines 11–12 compute the score associated with the extension of an existing sequence by action a. Finally, Lines 13–17 update $tables_G(a)$ based on information provided by the tuples in each $tables_G(v)$ such that v is an in-neighbor of a in any of the given PPGs.

The PPG_Insert algorithm works with a pruning algorithm PPG_Prune that updates the value of the *closed* attribute. For each (v, t) such that $v \in V_G$ and $t \in tables_G(v)$, PPG_Prune sets $t.closed$ to *true* iff $t.score \cdot maxP < \tau$ where

- $maxP = max_{x|(v,x)\in E}[(1 - \delta_G(v, x, t.A)) \cdot (1 - \rho_G(v, x, t.A))^z]$

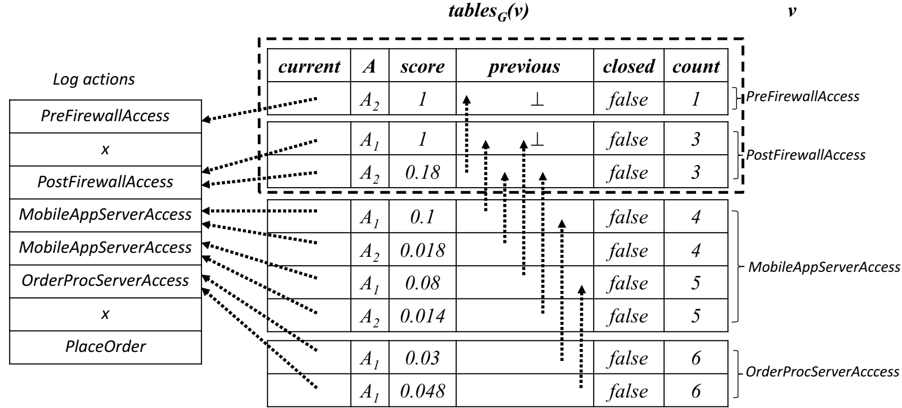
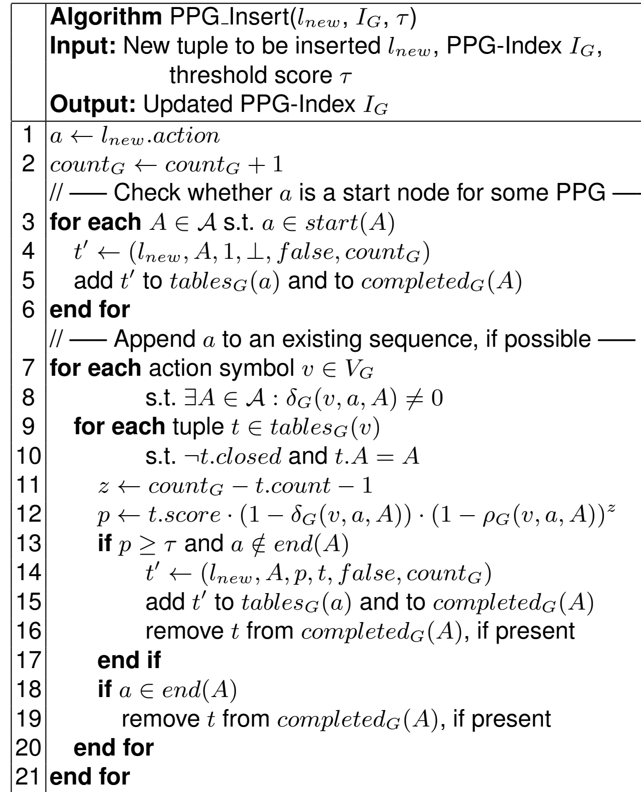


Fig. 4.5. Sequence of actions and status of $tables_G$.



Algorithm PPG_Insert(l_{new}, I_G, τ)
Input: New tuple to be inserted l_{new}, PPG-Index I_G,
threshold score τ
Output: Updated PPG-Index I_G

1 $a \leftarrow l_{new}.action$
2 $count_G \leftarrow count_G + 1$
// —— Check whether a is a start node for some PPG ——
3 **for each** $A \in \mathcal{A}$ s.t. $a \in start(A)$
4 $t' \leftarrow (l_{new}, A, 1, \perp, false, count_G)$
5 add t' to $tables_G(a)$ and to $completed_G(A)$
6 **end for**
// —— Append a to an existing sequence, if possible ——
7 **for each** action symbol $v \in V_G$
8 s.t. $\exists A \in \mathcal{A} : \delta_G(v, a, A) \neq 0$
9 **for each** tuple $t \in tables_G(v)$
10 s.t. $\neg t.closed$ and $t.A = A$
11 $z \leftarrow count_G - t.count - 1$
12 $p \leftarrow t.score \cdot (1 - \delta_G(v, a, A)) \cdot (1 - \rho_G(v, a, A))^z$
13 **if** $p \geq \tau$ and $a \notin end(A)$
14 $t' \leftarrow (l_{new}, A, p, t, false, count_G)$
15 add t' to $tables_G(a)$ and to $completed_G(A)$
16 remove t from $completed_G(A)$, if present
17 **end if**
18 **if** $a \in end(A)$
19 remove t from $completed_G(A)$, if present
20 **end for**
21 **end for**

Fig. 4.6. PPG_Insert and PPG_Retrieve algorithms.

- E is the set of edges of $t.A$
- $z = count_G - t.count - 1$

```
   Algorithm PPG_Retrieve(I_G)
   Input: PPG-Index I_G
            built using threshold τ
   Output: the set of all τ-unexplained
            situations for 𝒜 = {A_1, ..., A_k}
1  for each A_i ∈ 𝒜
2    U_i ← ∅
3    for each t ∈ completed_G(A_i)
4      l_e ← t.current
5      while t.previous ≠ ⊥
6        t ← t.previous
7      end while
8      l_s ← t.current
9      add (subLog(L, l_s, l_e)) to U_i
10   end for
11 end for
12 return maxComSubseq(U_1, ..., U_k)
```

Fig. 4.7. PPG_Retrieve algorithm.

Example 4.17. Consider a log whose associated sequence of action symbols is

$$\langle \text{PreFirewallAccess}, \text{x}, \text{PostFirewallAccess}, \text{MobileAppServerAccess},$$

$$\text{MobileAppServerAccess}, \text{OrderProcessingServerAccess}, \text{x}, \text{PlaceOrder} \rangle$$

The status of the PPG-Index after indexing this log is shown in Fig. 4.5.

The PPG_Retrieve algorithm, shown in Fig. 4.7, uses the Super-PPG to return the set of τ-unexplained situations for $\mathcal{A}$. For each $A \in \mathcal{A}$, PPG_Retrieve computes the set of τ-unexplained situations for A by (i) retrieving the index tuples pointed by $completed_G(A)$ and then (ii) following *previous* pointers until $previous = \perp$ for each retrieved index tuple, while storing the needed log tuples (*current* attribute). Finally, it returns the maximal common contiguous subsequences of the computed sets. The following result establishes correctness and completeness of the algorithms.

Proposition 4.18. *Consider a set $\mathcal{A}$ of PPGs, a log $L = l_1, \ldots, l_n$, and a threshold $\tau \in [0,1]$. Let I_G^i be the PPG-Index returned by PPG_Insert(l_i, I_G^{i-1}, τ). Then:*

$$I_G^i = \textit{PPG_Insert}(l_i, \textit{PPG_Prune}(I_G^{i-1}, \tau), \tau).$$

Moreover, PPG_Retrieve(I_G^n) is the set of all τ-unexplained situations for $\mathcal{A}$.

Proof. We start by showing that the correctness of Line 10 of PPG_Insert, which ignores those tuples in I_G^{i-1} whose *closed* attribute is *true*, is not affected by the application of PPG_Prune to I_G^{i-1}. PPG_Prune sets $t.closed = true$

when t points to a log tuple with associated action v that end a sequence $L^* \subseteq L$ representing an unexplained situation w.r.t. a PPG $t.A$ with score $t.score$. It is easy to see that the score of any sequence extending L^* cannot exceed $t.score \cdot maxP$, with $maxP$ being the maximum possible value of

$$(1 - \delta_G(e, t.A)) \cdot (1 - \rho_G(e, t.A))^z$$

where e is an outgoing edge of v in the Super-PPG and z is the number of noise actions encountered after indexing t (i.e., $z = count_G - t.count - 1$). As a consequence, based on Definition 4.4, if $t.score \cdot maxP < \tau$, L^* cannot be further extended.

The PPG_Insert algorithm indexes tuple l_i with associated action $l_i.action = a$. If $a \in start(A)$ for some $A \in \mathcal{A}$, the index tuple t' (with $t'.current = l_i$) that is added to $tables_G(a)$ must have no *previous* pointer, and its score must be set to 1 by Definition 4.4. Moreover, t' by itself represents an unexplained situation, so it is correctly added to $completed_G(A)$. If some $A \in \mathcal{A}$ has an edge from v to a, then the sequence obtained by adding l_i to the sequence represented by any index tuple t in $tables_G(v)$ has a score equal to $t.score \cdot (1 - \delta_G(v, a, A)) \cdot (1 - \rho_G(v, a, A))^z$, with $z = count_G - t.count - 1$. If this score is above τ and $a \notin end(A)$, then it is correct (i) to extend the existing sequence with l_i (obviously, we have $t'.previous = t$ in this case) and (ii) to remove t from $completed_G(A)$ based on Condition 4 of Definition 4.4, as it no longer represents an unexplained situation. Finally, if $a \in end(A)$, then it is correct to remove t from $completed_G(A)$ based on Condition 3 of Definition 4.4.

Finally, the correctness of the PPG_Retrieve algorithm immediately follows from the correctness of PPG_Insert and PPG_Prune. In fact, PPG_Retrieve reconstructs all unexplained situations for any PPG A_i by just following backward pointers in $table_G$. It is easy to see that, at the end of each iteration, t_s and t_e are the start and end log tuples of an unexplained situation for A_i—the set of all τ-unexplained situations is then the maximal common subsequence of such unexplained situations by Definition 4.9.

4.4.3 Example: updating and pruning a PPG-Index

Consider the log of Example 4.17, whose associated sequence of action symbols is

⟨PreFirewallAccess, x, PostFirewallAccess, MobileAppServerAccess,

MobileAppServerAccess, OrderProcessingServerAccess, x, PlaceOrder⟩

Assume the first three log tuples are already indexed as in Example 4.16, and that we execute PPG_Insert on the remaining tuples with $\tau = 10^{-3}$. When the first MobileAppServerAccess tuple is handled by PPG_Insert, two index tuples are added to $tables_G$(MobileAppServerAccess) (Fig. 4.5). The first one completes the sequence ⟨PostFirewallAccess, MobileAppServerAccess⟩ for

PPG A_1, so the score decreases by $1-\delta_G$(PostFirewallAccess, MobileAppServer-Access, A_1) w.r.t. its *previous* index tuple. The second one completes the sequence ⟨PreFirewallAccess, PostFirewallAccess, MobileAppServerAccess⟩ for PPG A_2, so the score decreases by $1-\delta_G$(PostFirewallAccess, MobileAppServer-Access, A_2). In both cases, on Lines 7–18 we have $v =$ PostFirewallAccess, $a =$ MobileAppServerAccess, and $z = 0$ because there are no log tuples between PostFirewallAccess and MobileAppServerAccess. The execution of PPG_Insert on the second MobileAppServerAccess tuple produces two additional tuples in $tables_G$(MobileAppServerAccess). The *score* values for these tuples are further decreased using the penalties associated with the (PostFirewallAccess, MobileAppServerAccess) edges in A_1 and A_2, respectively. This is due to the MobileAppServerAccess log tuple between PostFirewallAccess and MobileAppServer-Access, that must be now interpreted as noise—in this case, $z = 1$. After indexing the remaining log tuples, the situation of the tables is that of Fig. 4.5. Note that the PlaceOrder log tuple completed an occurrence of A_2 that extended all of the unexplained situations for A_2. Thus, in order to satisfy Condition 3 in Definition 4.3, PPG_Insert (Line 16) removed all the corresponding index tuples from $completed_G(A_2)$. $completed_G(A_1)$ contains instead two pointers to the tuples in $tables_G$(OrderProcessingServerAccess), and we have $count_G = 8$.

Assume now that PPG_Prune is executed on the index and consider tuple $t_{eb} \in tables_G$(PreFirewallAccess). We have $z = count_G - t_{eb}.count - 1 = 6$, so moving from PreFirewallAccess to PostFirewallAccess in A_2 would now make the score decrease by $(1-0.1)\cdot(1-0.8)^6 = 5.8\cdot 10^{-5}$. In the case of PlaceOrder, the score would instead decrease by $(1-0.9)\cdot(1-0.7)^6 = 7.3\cdot 10^{-5}$. Thus, we have $maxP = 7.3\cdot 10^{-5}$. Since $t_{eb}.score \cdot maxP < \tau$, we know that t_{eb} can no longer be linked to a new tuple. As a consequence, PPG_Prune sets $t_{eb}.closed$ to *true*, and t_{eb} is no longer considered by PPG_Insert (at Line 8).

4.5 Partitioning Super-PPGs Across a Compute Cluster

We implement the PPG-Index on a cluster of $(K+1)$ nodes in two steps. We propose 5 ways of partitioning the Super-PPG into K parts in a way that tries to minimize the expected inter-node communication. Each compute node is allocated one of these parts—the one remaining node is a master node. Within a compute node, a PPG-Index for the portion of the Super-PPG allocated to it is constructed. Each compute node also has a handoff protocol that governs inter-node communications—this will be discussed in the next section.

Let $\mathcal{A}$ be a set of PPGs and $G(\mathcal{A}) = (V_G, E_G, \delta_G, \rho_G)$ the corresponding Super-PPG. A vertex partition of $G(\mathcal{A})$ is a set of graphs

$$\mathcal{G} = \{G_1 = (V_1, E_1), \ldots, G_K = (V_K, E_K)\}$$

such that

- $\{V_1, \ldots, V_K\}$ is a partition of V_G

- $E_i = \{(v, v') \in E_G | \ v, v' \in V_i\}$

Given an edge $e = (v_i, v_j)$, an *edge cost function cost*(e) can be defined in different ways. We introduce different options in the rest of this section. These cost functions employ probability and noise degradation functions that consider the average of the probability and degradation values appearing on e across all the PPGs, that is, if $e = (v_i, v_j)$, then we define:

- $\delta'(e) = avg\{\delta_G(v_i, v_j, A) \mid A \in \mathcal{A}, \delta_G(v_i, v_j, A) > 0\}$
- $\rho'(e) = avg\{\rho_G(v_i, v_j, A) \mid A \in \mathcal{A}, \delta_G(v_i, v_j, A) > 0\}$

The *cost* of $\mathcal{G}$ is

$$\mathsf{CostD}(\mathcal{G}) = \sum_{1 \le i,j \le K \wedge i \ne j} \mathsf{CostD}(G_i, G_j)$$

where

$$\mathsf{CostD}(G_i, G_j) = \sum_{e=(v_i,v_j) \in E_G, v_i \in V_i, v_j \in V_j} cost(e)$$

We will try to find a $\mathcal{G}$ that minimizes $\mathsf{CostD}(\mathcal{G})$. This can be computed through a standard minimum cut algorithm such as [101].

4.5.1 Probability Partitioning (PP) and Probability-Penalty Partitioning (PPP)

The first two cost functions set the cost of an edge in terms of probability alone and in terms of both probability and penalty: $\mathsf{CostD}_{PP}(e) = 1 - \delta'(e)$ and $\mathsf{CostD}_{PPP}(e) = (1 - \delta'(e)) \cdot (1 - \rho'(e))$. Intuitively, CostD_{PP} tries to keep edges with a high transition probability on the same compute node. Likewise, CostD_{PPP} tries to keep edges with both a high transition probability and a high noise degradation value together on the same compute node. This is because the higher these values, the higher the probability that the two actions will be connected in an unexplained situation with a score above the threshold.

Example 4.19. Consider the Super-PPG of Example 4.14 (cf. Fig. 4.4). Suppose we have two compute nodes. A possible vertex partition is the one consisting of two graphs: G_1 containing vertices

$$\{\mathsf{PostFirewallAccess}, \mathsf{MobileAppServerAccess}, \mathsf{PreFirewallAccess}, \mathsf{PlaceOrder}\}$$

and G_2 cointaining vertices

$$\{\mathsf{MobileAppDBAccess}, \mathsf{OrderProcessingServerAccess}, \mathsf{CentralDBServerAccess}\}$$

The edges of the Super-PPG that go from one of the two graphs to the other are

- $e_1 =$ (PostFirewallAccess, CentralDBServerAccess)
- $e_2 =$ (MobileAppServerAccess, MobileAppDBAccess)
- $e_3 =$ (MobileAppServerAccess, OrderProcessingServerAccess)
- $e_4 =$ (MobileAppServerAccess, CentralDBServerAccess)

Moreover, we have

- $\delta'(e_1) = 0.1$, $\rho'(e_1) = 0.4$
- $\delta'(e_2) = 0.4$, $\rho'(e_2) = 0.3$
- $\delta'(e_3) = 0.4$, $\rho'(e_3) = 0.5$
- $\delta'(e_4) = 0.2$, and $\rho'(e_4) = 0.7$

The following costs are obtained by considering the probability partitioning:

- $\mathsf{CostD}_{PP}(e_1) = 1 - 0.1 = 0.9$
- $\mathsf{CostD}_{PP}(e_2) = 1 - 0.4 = 0.6$
- $\mathsf{CostD}_{PP}(e_3) = 1 - 0.4 = 0.6$
- $\mathsf{CostD}_{PP}(e_4) = 1 - 0.2 = 0.8$

Thus, the overall cost of the partition is $0.9 + 0.6 + 0.6 + 0.8 = 2.9$. On the other hand, the costs obtained via the probability-penalty partitioning are:

- $\mathsf{CostD}_{PPP}(e_1) = (1 - 0.1) \cdot (1 - 0.4) = 0.54$
- $\mathsf{CostD}_{PPP}(e_2) = (1 - 0.4) \cdot (1 - 0.3) = 0.42$
- $\mathsf{CostD}_{PPP}(e_3) = (1 - 0.4) \cdot (1 - 0.5) = 0.3$
- $\mathsf{CostD}_{PPP}(e_4) = (1 - 0.2) \cdot (1 - 0.7) = 0.24$

In this case, the overall cost of the partition is $0.54 + 0.42 + 0.3 + 0.24 = 1.5$.

4.5.2 Expected Penalty Partitioning (EPP)

Suppose $occ(v_i, v_j)$ is the expected number of log tuples appearing between two tuples with actions v_i and v_j in the log. The function *occ* can be easily learned from a historical log. We can then define

$$\mathsf{CostD}_{EPP}(e) = (1 - \rho'(e))^{occ(v_i, v_j)}$$

Thus, CostD_{EPP} assigns to an edge, a cost that takes into account the expected penalty value between the two actions involved in the edge. Again, the higher this value, the higher the probability that the two actions will be connected in an unexplained situation with a high score.

Example 4.20. Consider the vertex partition of Example 4.19 and assume function *occ* is defined as follows:

- *occ*(PostFirewallAccess, CentralDBServerAccess)=3
- *occ*(MobileAppServerAccess, MobileAppDBAccess)=4
- *occ*(MobileAppServerAccess, OrderProcessingServerAccess)=5.7
- *occ*(MobileAppServerAccess, CentralDBServerAccess)=2

The following costs are obtained by considering the expected penalty partitioning:

- $\mathsf{CostD}_{EPP}(e_1) = (1-0.4)^3 = 0.216$
- $\mathsf{CostD}_{EPP}(e_2) = (1-0.3)^4 = 0.2401$
- $\mathsf{CostD}_{EPP}(e_3) = (1-0.5)^{5.7} = 0.019$
- $\mathsf{CostD}_{EPP}(e_4) = (1-0.7)^2 = 0.09$

Thus, the overall cost of the partition is $0.216+0.2401+0.019+0.09 = 0.5651$.

4.5.3 Temporally Discounted Expected Penalty Partitioning (tEPP)

Given a log $L = l_1, \ldots, l_{|L|}$, we consider temporal windows of length t and discount for old occurrences. We define $occ_t(v_i, v_j, w)$ as the expected number of log tuples appearing between two tuples with actions v_i and v_j *in the w-th temporal window in the log.* Temporal windows are ordered starting from the end of the log, i.e., tuple t_k is in the w-th temporal window if

$$|L| - w \cdot t + 1 \leq k \leq |L| - (w-1) \cdot t$$

Then we consider a discount factor $d \in [0,1]$ and define

$$\mathsf{CostD}_{tEPP}(e) = \sum_{w>0, occ_t(v_i,v_j,w)>0} (1-\rho'(e))^{\frac{1}{d^w} occ_t(v_i,v_j,w)}$$

Example 4.21. Consider the vertex partition of Example 4.19. Suppose we have a log with 800 tuples and we want to consider temporal windows of length 500. This means that there are two temporal windows to be considered: one goes from tuple 301 to tuple 800, while the other goes from tuple 1 to tuple 300. Suppose the discount factor $d = 2$ and occ is defined as follows:

- occ(PostFirewallAccess, CentralDBServerAccess, $1) = 3$
- occ(PostFirewallAccess, CentralDBServerAccess, $2) = 2$
- occ(MobileAppServerAccess, MobileAppDBAccess, $1) = 4$
- occ(MobileAppServerAccess, MobileAppDBAccess, $2) = 1$
- occ(MobileAppServerAccess, OrderProcessingServerAccess, $1) = 5.7$
- occ(MobileAppServerAccess, OrderProcessingServerAccess, $2) = 4$
- occ(MobileAppServerAccess, CentralDBServerAccess, $1) = 2$
- occ(MobileAppServerAccess, CentralDBServerAccess, $2) = 3.3$

The following costs are obtained by considering the temporally discounted expected penalty partitioning:

- $\mathsf{CostD}_{tEPP}(e_1) = (1-0.4)^{0.5\cdot 3} + (1-0.4)^{0.25\cdot 2} = 1.239$
- $\mathsf{CostD}_{tEPP}(e_2) = (1-0.3)^{0.5\cdot 4} + (1-0.3)^{0.25\cdot 1} = 1.405$
- $\mathsf{CostD}_{tEPP}(e_3) = (1-0.5)^{0.5\cdot 5.7} + (1-0.5)^{0.25\cdot 4} = 0.639$
- $\mathsf{CostD}_{tEPP}(e_4) = (1-0.7)^{0.5\cdot 2} + (1-0.7)^{0.25\cdot 3.3} = 0.670$

Thus, the overall cost of the partition is 3.953.

4.5.4 Occurrence Partitioning (OP)

In this case, we consider a pruning threshold c and define

$$\mathsf{CostD}_{OP}(e) = \frac{1}{occ(v_i, v_j)} = \begin{cases} \frac{1}{occ(v_i,v_j)} & \text{if } \frac{1}{occ(v_i,v_j)} \geq c; \\ 0 & \text{otherwise} \end{cases}$$

Thus, CostD_{OP} is inversely proportional the expected number of log tuples appearing between two tuples with actions v_i and v_j. However, if this ratio is too small (i.e., below threshold c), we assume there is no cost to placing v_i, v_j on different nodes.

Example 4.22. Consider the partition of Example 4.19 and suppose occ is defined as in Example 4.20. Let $c = 0.3$. Then: $\mathsf{CostD}_{OP}(e_1) = 0.33$, $\mathsf{CostD}_{OP}(e_2) = 0$, $\mathsf{CostD}_{OP}(e_3) = 0$, and $\mathsf{CostD}_{OP}(e_4) = 0.5$. Thus, the overall cost of the partition is 0.83 .

4.6 Parallel Detection

In this section we show how the PPG-Index and the PPG_Retrieve algorithm can be adapted to a cluster of K compute nodes and a master node — the required modifications involve coordination through inter-node communication.

After partitioning a Super-PPG $G(\mathcal{A}) = (V_G, E_G, \delta_G, \rho_G)$ into K components, each component is assigned to a different compute node—thus, each vertex $v \in V_G$ is managed by a single compute node denoted $node(v)$. All compute nodes store the $tables_G$ structures for the vertices they manage, and their own $completed_G$ structures. However, each tuple t in $tables_G$ has a seventh component, called *starting*, i.e. the ID of the log tuple from which the (partial) unexplained situation represented by t started. All nodes in the cluster also store a copy of $G(\mathcal{A})$ and the value of the threshold τ.

The master node, in order to index a new log tuple l_{new}, sends it to the compute node that manages $l_{new}.action$ after updating the value of $count_G$ (pPPG_Send algorithm, reported in Fig. 4.8). At any time, the master node can execute the pPPG_Retrieve algorithm (Fig. 4.8) to build the set of τ-unexplained situations detected so far. Note that the only difference between the pPPG_Retrieve algorithm and its single-CPU counterpart PPG_Retrieve is that, for each PPG A_i, instead of retrieving tuples from the local $completed_G(A_i)$, the pPPG_Retrieve algorithm queries compute nodes—then, instead of following backward pointers through the PPG-Index, it can directly add the corresponding sub-logs to the set U_i by using the *starting* component of the index tuples.

The compute nodes execute the pPPG_Insert algorithm (reported in Fig. 4.9) when requested by the master node, and communicate with one another through the pPPG_Get algorithm (Fig. 4.9). In particular, Lines 19–22 of the pPPG_Insert algorithm retrieve from another node a set T of index

	Algorithm pPPG_Send(l_{new}) **Input:** New log tuple l_{new}
1	$count_G \leftarrow count_G + 1$
2	$node(l_{new}.action)$.pPPG_Insert($l_{new}, count_G$)

	Algorithm pPPG_Retrieve() **Output:** τ-unexplained situations for $\mathcal{A} = \{A_1, \ldots, A_k\}$
1	**for each** $A_i \in \mathcal{A}$
2	$U_i \leftarrow \emptyset$
3	**for each** compute node $node$ managing a vertex in A_i
4	get $completed_G(A_i)$ from $node$
5	**for each** $t \in completed_G(A_i)$
6	add (subLog($L, t.starting, t.current$)) to U_i
7	**end for**
8	**end for**
9	**end for**
10	**return** maxCommSubseq($U_1, \ldots, U_k$)

Fig. 4.8. Algorithms executed by the master node.

tuples that represent unexplained situations that can be extended, and update the local $tables_G$ and $completed_G$. The pPPG_Get algorithm corresponds to Lines 8–16 of PPG_Retrieve — in this case, the set T is obviously returned to the requesting node. All compute nodes run the PPG_Prune algorithm given in Section 4.4 as well, independently and concurrently with pPPG_Insert and pPPG_Get.

4.7 Experimental Results

The Super-PPG management/partitioning and detection functionalities were developed in C++ while the parallel detection algorithms were implemented using the Message Passing Interface parallel programming model. We used METIS[3] libraries for partitioning. Super-PPGs with several hundreds of vertices were always partitioned in a few milliseconds. We tested PADUA's accuracy and processing time using the *SCOPE*[4] distributed computing infrastructure at the University of Naples. SCOPE consists of over 300 compute nodes (quad-core Intel Xeon 2.33GHz processors with 8GB RAM) communicating by dedicated fiber channels. Tests were done with both video surveillance and network traffic data.

[3] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`
[4] `www.scope.unina.it`

	Algorithm pPPG_Insert(l_{new}, $count_G$) **Input:** New tuple to be inserted l_{new}, tuple count $count_G$
1	$a \leftarrow l_{new}.action$
	// —– Check whether a is a start node —–
2	**for each** $A \in \mathcal{A}$ s.t. $a \in start(A)$
3	$t' \leftarrow (l_{new}, A, 1, \perp, false, count_G, l_{new})$
4	add t' to $tables_G(a)$ and to $completed_G(A)$
5	**end for**
	// —– Append a to an existing sequence, if possible —–
6	**for each** action symbol $v \in V_G$ s.t. $\exists A \in \mathcal{A} : \delta_G(v, a, A) \neq 0$
7	**if** $node(v)$ is the local node
8–17	// —– Same as PPG_Insert (lines 8–17) —–
18	**else** // —– In this case, coordinate with another cluster node —–
19	**for each** tuple $t \in node(v)$.pPPG_Get($v, a, A, count_G$)
20	$t' \leftarrow (l_{new}, A, p, t, false, count_G, t.starting)$
21	add t' to $tables_G(a)$ and to $completed_G(A)$
22	**end for**
23	**end if**
24	**end for**

	Algorithm pPPG_Get($v, a, A, count_G$) **Input:** Action symbols v, a, PPG A, tuple count $count_G$ **Output:** Set T of index tuples
1	$T \leftarrow \emptyset$
2	**for each** tuple $t \in tables_G(v)$ s.t. $\neg t.closed$ and $t.A = A$
3	$z \leftarrow count_G - t.count - 1$
4	$p \leftarrow t.score \cdot (1 - \delta_G(v, a, A)) \cdot (1 - \rho_G(v, a, A))^z$
5	**if** $p \geq \tau$ and $a \notin end(A)$
6	add t to T
7	remove t from $completed_G(A)$, if present
8	**if** $a \in end(A)$ **then** remove t from $completed_G(A)$, if present
9	**end for**
10	**return** T

Fig. 4.9. Algorithms executed by the compute nodes.

4.7.1 Video Surveillance Domain

PADUA was tested on the video surveillance experimental setup and measures used in [78]. The log was generated using the ITEA CANDELA video dataset[5] which contains several staged package exchanges and object drop-offs and pick-ups. 64 different action symbols were defined in a semi-automatic way using both image processing libraries and human annotation.

[5] http://www.multitel.be/image/research-development/research-projects/candela/

Result Quality

Precision and recall were evaluated against ground truth provided in [78] by human annotators who were given a set of known activity descriptions along with graphical representation of the PPGs (we used 30 PPGs), and then asked to watch the video and identify video segments which where unexplained. Precision P and recall R were defined as

$$P = \frac{|\{L_u^a \text{ in } U^a | \exists L_u^h \text{ in } U^h \text{ s.t. } L_u^a \simeq L_u^h\}|}{|U^a|}$$

$$R = \frac{|\{L_u^h \text{ in } U^h | \exists L_u^a \text{ in } U^a \text{ s.t. } L_u^a \simeq L_u^h\}|}{|U^h|}$$

where U^a is the set of unexplained situations returned by the algorithm, U^h is the set of sequences identified as unexplained by human annotators, and $L_u^a \simeq L_u^h$ iff L_u^a and L_u^h overlap by a percentage not smaller than 75%. Fig. 4.10 shows the results obtained.

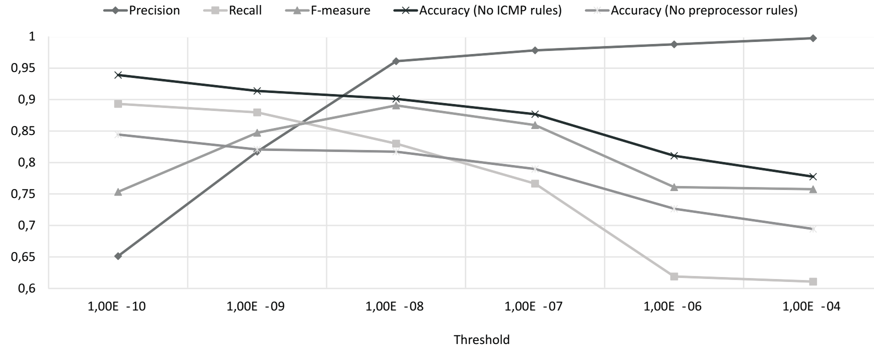


Fig. 4.10. Precision, recall, and F-measure for the video surveillance dataset, and accuracy for the cybersecurity dataset.

The τ-value that yielded the highest *F-measure*[6], as well as the highest recall, was 10^{-8}. PADUA's precision, recall, and F-measure, respectively, were 0.96, 0.83, and 0.89. In contrast, the corresponding values obtained with the best possible parameter settings in [78] were $0.73, 0.72$, and 0.72—significantly lower in all respects than PADUA. Moreover, the experiments confirm the claim in Proposition 4.10: with higher values of τ, the average length of unexplained situations decreases and, as a consequence, we obtain better precision and worse recall.

[6] F-measure is given by $\frac{2PR}{P+R}$.

Processing Times

To evaluate PADUA's scalability when using each of our 5 partitioning schemes, we fixed τ to the value that maximized the F-measure and measured how processing times varied as we varied the length of the video and the number of compute nodes. Figs. 4.11 and 4.12(dark line) show the results obtained.

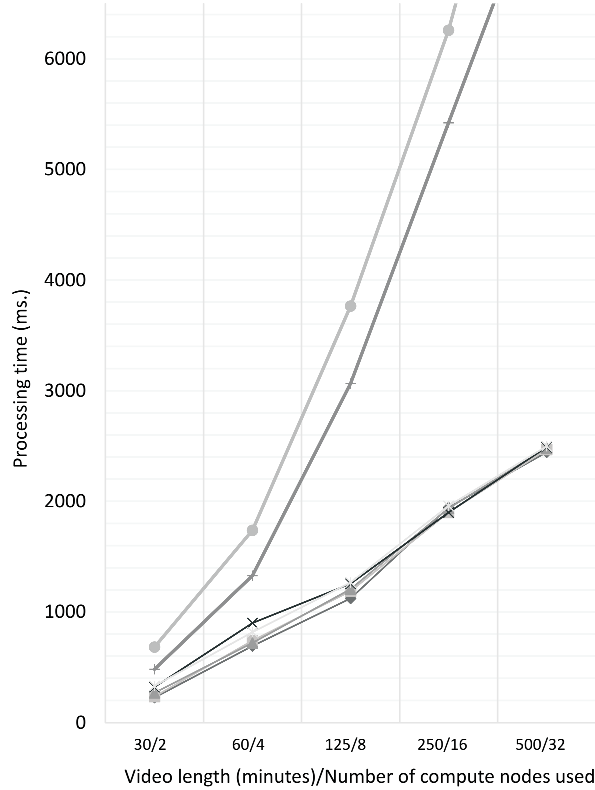


Fig. 4.11. Processing times for the video surveillance dataset.

The results show that PADUA provides extremely good performance and scalability. It is able to process up to 294K tuples per second on the longest video sequence (each second of video corresponds to 25 log tuples). Moreover, a 16x increase in the log size only results in a 10x increase in processing time. Though the different partitioning schemes show similar performance for long video sequences, OP, PP, and PPP show a performance advantage for video lengths of up to 125 minutes.

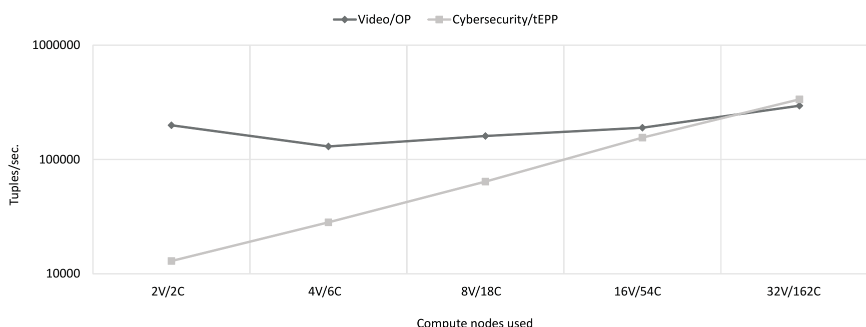


Fig. 4.12. Tuples processed per second for two dataset/cost function combinations. Label xV/yC on the x-axis stands for "x nodes for the video surveillance dataset, y nodes for the cybersecurity dataset". The video/traffic lengths considered are those corresponding to x nodes in Fig. 4.11 and y nodes in Fig. 4.13.

Finally, PADUA clearly outperforms the approach in [78] — for instance, their processing times for 500 minutes of video were around 10^5 ms. This essentially due to (*i*) the fact that [78] considers possible worlds and there are exponentially many of them, while we do not, (*ii*) we use a specifically designed index, and (*iii*) the efficiency of the parallel algorithms.

4.7.2 Cybersecurity Domain

The Naples Network Traffic dataset is built based on a *Network Sniffer*, a *Network Intrusion Detection System* (IDS), and an *Alert Aggregation* component. The Sniffer (implemented by Wireshark[7]) captures network traffic, whereas the IDS (implemented via *SNORT*[8]) analyzes such traffic and generates a sequence of action symbols. As the IDS may return lots of alerts, the Alert Aggregation module aggregates multiple alerts triggered by the same action into a macro-alert based on a set of aggregation rules. The dataset contained 2 full days of traffic (about 1,215,000 log tuples). We defined 350 PPGs, corresponding to the available SNORT rules, containing 722 action symbols. The set of SNORT rules comprised ICMP rules, designed to analyze ICMP packets, and preprocessing rules that handle situations where packets have to be decoded into plain text for the actual SNORT rules to trigger.

Result Quality

In the cybersecurity domain, we measured the accuracy of the results as follows. First, we detected all occurrences of the set of SNORT rules in the log. Then we ignored a certain subset of the rules and identified the unexplained

[7] http://www.wireshark.org/
[8] http://www.snort.org/

situations. Occurrences of ignored PPGs were expected to have a relatively high probability of being unexplained situations, as there is no model for them. We measured the fraction of such occurrences that were correctly flagged as unexplained for different values of τ. Specifically, we considered two settings: one where only ICMP rules were ignored, and another where only preprocessor rules were ignored from a single IP. The results for a log containing 135K tuples, corresponding to 270 minutes of network traffic, are reported in Fig. 4.10.

The results show good accuracy values. When ICMP rules were ignored, sequences where ICMP activities were occurring were flagged as unexplained situations in the majority of cases—the same happened when preprocessor rules were ignored. As expected, the better accuracy obtained with lower τ-values corresponded to higher processing times: the full log was processed in 1,875 ms. with $\tau = 10^{-4}$, 2,345 ms. with $\tau = 10^{-8}$, and 3,502 ms. with $\tau = 10^{-10}$.

Processing Times

As in the video surveillance case, we measured how processing times varied with length of the network traffic, using an increasing number of compute nodes, with $\tau = 10^{-8}$. Figs. 4.12(light line) and 4.13 show the results obtained.

The results confirm that PADUA is able to process up to 335K tuples per second on the largest network traffic length. More importantly, an 81x increase in the traffic length only results in a 3x increase in processing time. Moreover, the different partitioning schemes show similar performance for shorter traffic lengths—for long traffic lengths, EPP and tEPP appear to be the best schemes. This suggests OP, OPP, and PPP to be better options when dealing with smaller logs and fewer action symbols (as in the case of the video surveillance domain), while EPP and tEPP appear better suited for larger logs and more action symbols (resulting in larger Super-PPGs).

4.8 Conclusions

To the best of our knowledge, PADUA is the only parallel approach to date for identifying unexplained situations in historical transaction logs which occur naturally in many domains. Starting with the same activity model as [78] (extended slightly to incorporate penalties via the notion of a probabilistic penalty graph or PPG), we showed how we can merge a set of PPGs into a single Super-PPG and provided the PPG-Index data structure to store, update, and analyze observations as they come in (e.g., from a video surveillance application or a cybersecurity application). This Super-PPG can then be "split" across K compute nodes in a cluster of $(K + 1)$ nodes in many ways—we present 5 such ways of doing so. The resulting partitioned graph has one

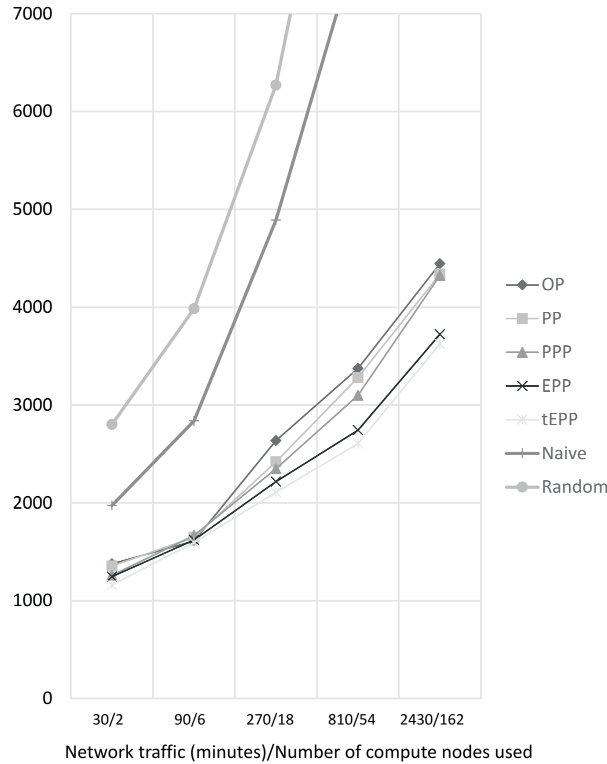


Fig. 4.13. Processing times for the cybersecurity dataset.

part stored on each of K compute nodes with one node serving as a master node. We develop parallel algorithms that achieve coordination amongst these different compute nodes so as to leverage parallelism when detecting unexplained situations.

Experimental results show great promise. On both video and cybersecurity datasets, we was able to significantly improve on past results for unexplained situation detection. Specifically, precision, recall and F-measure go up to 0.96, 0.83, 0.89 compared to 0.73, 0.72, and 0.72 respectively from past work—a significant improvement. Second, scaling is substantial. We was able to process 500 minutes of video (after image processing) on 32 compute nodes in under 2.5 seconds which is two orders of magnitude better than past work. On the cybersecurity dataset, we was able to process up to 335K observation (log) tuples per second.

Part III

Adversarial Defense

5

Pareto-Optimal Adversarial Defense of Enterprise Systems

The National Vulnerability Database (NVD) maintained by the US National Institute of Standards and Technology provides valuable information about vulnerabilities in popular software, as well as any patches available to address these vulnerabilities. Most enterprise security managers today simply patch the most dangerous vulnerabilities — an adversary can thus easily compromise an enterprise by using less important vulnerabilities to penetrate an enterprise. In this chapter, vulnerabilities in an enterprise are captured as a vulnerability dependency graph (VDG) and ti is showed that attacks graphs can be expressed in them. First we ask the question: what set of vulnerabilities should an attacker exploit in order to maximize his expected impact? We show that this problem can be solved as an integer linear program. The defender would obviously like to minimize the impact of the worst case attack mounted by the attacker — but the defender also has an obligation to ensure a high productivity within his enterprise. We propose an algorithm that finds a Pareto-optimal solution for the defender that allows him to simultaneously maximize productivity and minimize the cost of patching products on the enterprise network. We have implemented this framework and show that run-times of our computations are all within acceptable time bounds even for large VDGs containing 30K edges and that the balance between productivity and impact of attacks is also acceptable.

5.1 Introduction

Security managers working for large organizations face a formidable challenge in protecting the security of their enterprises. First, there is a huge array of software running on their servers. The software can be of varying quality from a security point of view — for instance, some software may come from reputed software firms like Microsoft and Oracle, while other pieces of software may constitute shareware or freeware downloaded as needed by users for various projects. The US National Institute of Standards and Technology (NIST) has

made a commendable effort to track tens of thousands of software components that are widely used and build a database describing various aspects of the vulnerabilities in these commercial software components. The result is a National Vulnerability Database[1] (NVD for short) that consists of over 50K vulnerabilities identified in software, together with information on the availability of patches for those machines and the impact that vulnerabilities could have on security if left un-patched.

As patching software is a task that takes time and effort (and people), harried enterprise security managers typically patch those vulnerabilities that pose the highest threat (see, for example, [102, 103]). This effectively leaves the door wide open for a semi-smart attacker who can simply try to guess the type of software that a large company probably has installed on its servers and then tries to penetrate the system through vulnerabilities that are not rated as high impact vulnerabilities.

In this chapter, we attempt to take a more strategic view of this situation. First, we ask the question: given a set of publicly known information such as the cost of patching vulnerabilities (which can be readily inferred by anyone), what is the strategy that an intelligent attacker would use in order to maximize the expected damage he can cause? With this in hand, the defender can come up with defensive strategies that minimize the expected damage the attacker can inflict. We allow the defender to do two things: (*i*) deactivate certain products (e.g. if they have serious vulnerabilities) which could reduce the impact of attacks and (*ii*) apply patches to certain vulnerabilities. The first method has a potential impact on productivity of the enterprise while the second has a time/cost implication. We define the optimal strategy of the defender as a Pareto optimization problem and show how to find the set of all optimal strategies for the defender. We derive a number of complexity results associated with the attacker's goal of finding an attack that maximizes his expected impact, and also the defender's goal of taking steps to minimize the maximal impact the attacker can have. We have implemented algorithms and tested them on 4 real-world vulnerability dependency graphs (a more general version of attack graphs). Results show that algorithms work in reasonable amounts of time on real world networks and provide options to enterprise security managers that represent different combinations of maximizing productivity and minimizing expected attack impact. This prototype implementation shows that run-times of our computations are all within acceptable time bounds even for large VDGs containing 30K edges and that the balance between productivity and impact of attacks is also acceptable.

The rest of this chapter is organized as follows. Section 5.2 contains a detailed description of related work in this area. Section 5.3 introduces the notion of vulnerability dependency graphs (VDGs). Section 5.4 describes the attacker's strategy and the defender's strategy as a two-person game and formally relates these strategies to VDGs. Section 5.5 performs a detailed Pareto

[1] nvd.nist.gov

analysis of the game from the defender's perspective by first formulating the game as a bi-objective optimization problem (maximize productivity and minimize impact). A detailed set of complexity results in this section describe the complexity of the defender's ability to compute an optimal strategy. It also formulates the problem of solving the bi-objective optimization problem as a mixed integer linear program (MILP) and proves the correctness of these MILPs. Section 5.6 describes the results of experiments we have conducted. The experiments show that the framework delivers Pareto-optimal solutions for the defender that provide a good balance between productivity of the enterprise and the impact of the attacker's attacks. Moreover, the algorithms run within reasonable time bounds even on large VDGs.

5.2 Related Work

NIST's NVD effort builds on Common Vulnerability Scoring System (CVSS) [104] and the Common Weakness Scoring System (CWSS) [105] to provide standard ways for security analysts and vendors to rank known vulnerabilities and software weaknesses using numerical scores.

A number of tools are available for scanning network vulnerabilities, such as Nessus [106], but these only report isolated vulnerabilities. By their very nature, these vulnerabilities are highly interdependent — a machine's susceptibility to attack depends on the vulnerabilities of the other machines in the network. An attack graph models these interdependencies by enumerating all possible sequences of vulnerabilities that attackers may exploit to reach a goal state [107, 108, 109, 110, 111, 112, 113]. An attack graph can be constructed either forward, starting from the initial state [109, 111] or backward from the goal state [25, 114]. Model checking was first used to analyze whether a goal state is reachable from the initial state [114], but later used to enumerate all possible sequences of attacks between the two states [25, 115]. Because algorithmic complexity of early attack graph formalisms was exponential, much of the subsequent research focused on scalability. Under reasonable assumptions, attack graph analysis can be formulated in logic, making it unnecessary to explicitly enumerate states. This leads to polynomial rather than exponential complexity [61]. In more recent work, attack graphs have been used for correlating intrusion alerts [22, 62] and to find the minimal set of exploits from which the goal state is reachable [61, 64, 116].

This framework uses Pareto optimization and game theory to help enterprise security officers determine what software to patch, given constraints on their resources. Though this has never been done previously (to the best of our knowledge), there has been very interesting work on the use of game theory for network security. Physical layer has considered eavesdropping (i.e. listening and analyzing data from the network without interacting with it) and jamming (i.e. attack that can disrupt data transmission).

Some past work focuses on security in the physical layer from eavesdropping and jamming attacks. Here, players include attackers, non malicious users (that use the physical layer), and the layer itself (with its access control policy). These games are largely based on performance indexes of the physical layer, and the main goal is to optimize these performance indexes. [117, 118, 119, 120] use game-theory to study jamming attacks, while [121] use Stackelberg games [122].

Game theory has also been used to analyze and design security protocols for self-organizing the networks. [123] uses a two-player zero sum game where the first player executes jamming attacks while the second player is a set of mobile nodes taking countermeasures. An example countermeasure would revoke the access to a network from a malicious user. [124, 125] use game theory to design revocation protocols for ephemeral networks. [126] studies sensor networks where an attacker can physically capture sensor nodes, replicate the nodes, deploy them into the network, and proceed to take over the network. A multi-player game is formalized in order to model the non-cooperative strategic behavior between the attackers and the network.

Game theory has also been used in intrusion detection systems (IDS). [127] use a zero-sum stochastic game to model the dynamic configuration of the IDS (the defender) in response to a sequence of attacks by the attacker. [128] considers a network of intrusion detection systems where the IDSs cooperate to improve the security of the network. The authors propose an incentive-compatible trust based resource allocation scheme. In this scheme, each IDS allocates “help” resources proportionally to the trustworthiness and the amount of help resource allocated by its neighbors. Moreover, it is proved with an n-person non-cooperative game that under certain conditions, there exists a unique Nash equilibrium (that represents global satisfaction) and a decentralized algorithm (running on each IDS) to reach this equilibrium is provided. [129] provides a more exhaustive survey of research in network security.

Finally, [130, 131, 132] consider the problem of finding plans for patching vulnerabilities, that are tradeoffs between cost and risk, by using the Pareto analysis. However, in these papers (i) the productivity can be considered as a cost, but it is not possible to consider cost and productivity separately; (ii) the risk is defined on acyclic structures representing attacks (our dependency vulnerability graphs may have cycles); (iii) the risk value does not consider the worst case. In [131] they also define a problem involving a game theory-based solution, but they do not mix this with the Pareto analysis and assume that the attacker does not know the strategy of the defender (instead we assume this is possible through network scans). My proposal encompasses all of the above important aspects.

5.3 Vulnerability Dependency Graphs

The goal of this section is to introduce the basic concept of a vulnerability dependency graph (VDG for short) which is rich enough to express all attack graphs (cf. Appendix A). Suppose $\mathbf{PR}$ is a finite set of software products[2] and suppose $\mathbf{V}$ is a set of vulnerabilities. We use $\mathsf{Vuln}(pr) \subseteq \mathbf{V}$ to denote the set of all vulnerabilities of a particular product $pr \in \mathbf{PR}$. Conversely, for any vulnerability $v \in \mathbf{V}$, let $PR(v)$ denote all the products in $\mathbf{PR}$ having that vulnerability. Thus, $pr \in PR(v)$ iff $v \in \mathsf{Vuln}(pr)$. For each vulnerability $v \in \mathbf{V}$ there is a set $PA(v)$ of patches that fix v. We assume that any patch in $PA(v)$ fixes v, i.e. only one patch needs to be applied to fix a vulnerability and that patch fully covers the vulnerability. We denote the set of all patches as $\mathbf{PA}$. Note that a patch may fix one or more vulnerabilities, but a vulnerability may have zero, one, or many patches.

Vulnerabilities may depend on one another. To exploit a vulnerability v_2, an attacker may first need to exploit another vulnerability v_1. This leads to the idea of a vulnerability dependency graph.

Definition 5.1 (Vulnerability dependency graph). *A* vulnerability dependency graph *(VDG for short) is a directed graph $G = (\mathbf{V}, E)$ where $\mathbf{V}$ is the set of vulnerabilities (vertices) and $E \subseteq \mathbf{V} \times \mathbf{V}$ is the set of edges.*

Intuitively, an edge from vulnerability v_1 to a vulnerability v_2 means that v_2 can be exploited if v_1 is exploited. A vulnerability with zero in-degree can be exploited directly.

Example 5.2. Fig. 5.1 shows a toy vulnerability dependency graph G with $\mathbf{V} = \{v_1, \ldots, v_7\}$, $\mathbf{PR} = \{pr_1, \ldots, pr_5\}$, $\mathbf{PA} = \{pa_1, \ldots, pa_7\}$.

For each vulnerability $v \in \mathbf{V}$, the values of $PR(v)$ and $PA(v)$ are depicted in the dotted boxes. For instance, the box at the top left of Fig. 5.1 states that vulnerability v_1 exists in products pr_1 and pr_2 and there are two patches for it, pa_1 and pa_2. Likewise, the impact of this vulnerability is 7.

Given a VDG $G = (V, E)$, We use $S(G)$ to denote the set of all vertices having in-degree 0, and $in(G, v)$ and $out(G, v)$, respectively, to denote the sets $\{v' \mid (v', v) \in E\}$ and $\{v' \mid (v, v') \in E\}$, respectively. In the case of Fig. 5.1, we have $S(G) = \{v_1, v_2\}$, $in(G, v_4) = \{v_1\}$, and $out(G, v_4) = \{v_5, v_6\}$.

Each vulnerability v has an impact $\mathsf{Impact}(v)$ which is a measure of the impact of v on an enterprise if left unpatched. $\mathsf{Impact}(v)$ can be estimated in many different ways from NIST's Commmon Vulnerability Scoring System (CVSS). For each vulnerability, CVSS describes an access vector (of how the vulnerability can be exploited), an access complexity that is a measure of how difficult it is to exploit a vulnerability, an authentication metric that measures how often an attacker must authenticate himself, a confidentiality metric that

[2] To consider different machines and product versions, we can define the elements in $\mathbf{PR}$ as $(machine, product, version)$ triples.

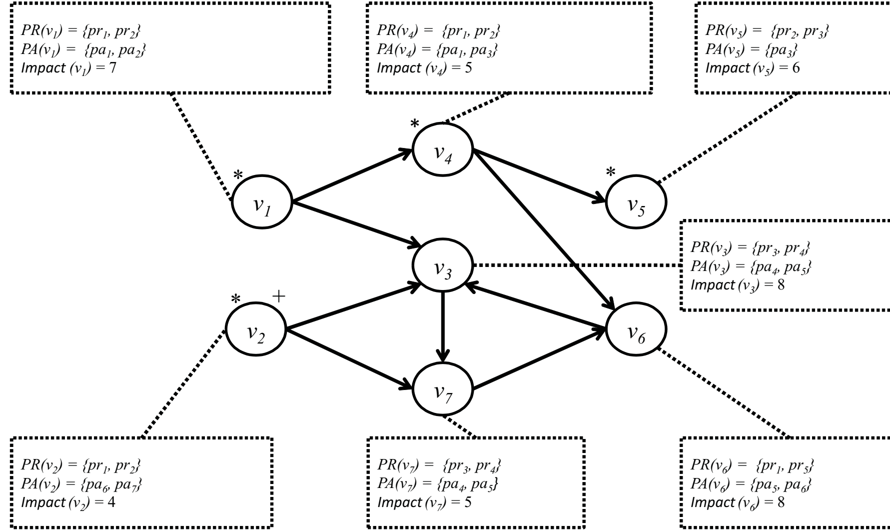


Fig. 5.1. Example vulnerability dependency graph

assesses the impact on confidentiality if the vulnerability is exploited, a similar metric relating to the impact on integrity of the system, and a metric describing the impact on the availability of the system if the vulnerability is exploited in an attack. Based on these parameters, NIST's CVSS describes an overall impact of the vulnerability. We do not take a position on whether NIST's measure of impact of a vulnerability is the correct one — rather, We use any function that measures impact of a vulnerability and merely point to CVSS to assert that at least one such "semi-standard" measure exists in the literature. The values of Impact for the vulnerabilities of Example 5.2 are depicted in dotted boxes in Fig. 5.1.

Each patch $pa \in PA$ has a cost for the enterprise as a whole, denoted by $\mathsf{CostD}(pa)$. This cost may denote many things, e.g. the time involved in applying the patch, the labor cost, the lost productivity if part of the system needs to be taken down to apply the patch, etc. In this chapter, we assume a single cost to apply a patch across the whole enterprise. The *total cost* of applying a set $PA \subseteq \mathbf{PA}$ of patches to an enterprise is defined as $Tc(PA) = \sum_{pa \in PA} \mathsf{CostD}(pa)$.

We also assume that each product $pr \in \mathbf{PR}$ has an associated productivity value $\mathsf{Prod}(pr)$ that represents the importance/usefulness of pr in the organization. The productivity value of a product may be captured in many ways, e.g. by monitoring how much the product was used per day (averaged over some time frame), by a user survey, by a weighted combination of the usage of the product per day and the ranks of the people using it, and so forth. The *total productivity of a set* $PR \subseteq \mathbf{PR}$ is simply

$$Tp(PR) = \sum_{pr \in PR} \mathsf{Prod}(pr).$$

Example 5.3. Suppose we return to Example 5.2 and set $\mathsf{CostD}(pa_i) = i$ and $\mathsf{Prod}(pr_i) = i$. Then $Tc(\mathbf{PA}) = 28$ and $Tp(\mathbf{PR}) = 15$.

Before concluding this section, we note that even though VDGs appear to be deterministic, they are in fact non-deterministic. For instance, consider the node v_3 in Fig. 5.1. In this figure, we see that for an attacker to exploit vulnerability v_3, he must first have exploited either vulnerability v_1 or v_2. This is a form of non-determinism as, by non-deterministically exploiting either of these two vulnerabilities, the attacker can exploit v_3. In general, for an attacker to exploit a vulnerability v in some arbitrary VDG, he must have previously exploited any vulnerability in $pred(v)$ where $pred(v)$ is the set of all predecessors of v in that VDG. One can have even more complex models of VDGs in which, for instance, we assign a probability to each edge in the VDG such that for all vertices v (that are not source vertices, i.e. vertices with in-degree 0), it is the case that $\Sigma_{(u,v) \in E} prob(u, v) = 1$. The sum constraint says that the sum of the probabilities of the edges incoming to any non-source vertex v must be one. For instance, in the case of Fig. 5.1, we might label the edge from v_1 to v_3 with 70% and the edge from v_2 to v_3 may be labeled with 30%. This may suggest that there is 70% likelihood that vertex v_3 will be exploited by an attacker who first exploits v_1 and that there is 30% probability that vertex v_3 will be exploited by an attacker who first exploits v_2. We did not include such probabilistic VDGs in this chapter for two reasons. First, the question arises of how to find these probabilities. One way to achieve this is to consider either historical attack sequence data (but we did not have such data) or to use CVSSs "access complexity" measure to subjectively assign such probabilities. Second, we wanted to consider worst case scenarios where attackers could use all possible paths in VDGs to launch their exploits — from a security perspective, it makes sense to take care of worst case scenarios. However, we believe the problem of studying probabilistic VDGs is an important one for future work.

5.3.1 From Attack Graphs to Vulnerability Dependency Graphs

Different *attack graph* (AG for short) formalisms are heavily used in the security literature [25, 107, 113, 114, 115]. In this section, we show that the VDG framework is rich enough to express all these attack graphs. We do this by providing a translation of AGs into VDGs — but first, we must define AGs.

Definition 5.4 (Attack graph). *An Attack Graph (AG for short) is represented by a tuple* $AG = < M, E, \mathsf{TPart}, V_M, V_E, v_M, v_E >$ *where*

- M *is a set of machines.*

- $\mathsf{TPart} = \{M_1, \ldots, M_k\}$ *is a partition of* M *i.e.* $M = \bigcup_{i=1}^{k} M_i$ *and for each* $M_1, M_2 \in \mathsf{TPart}$, $M_1 \cap M_2 \neq \emptyset$. TPart *is called a "trust partition" and each* $M_i \in \mathsf{TPart}$ *is called a trust component.*
- $E \subseteq \bigcup_{M_i, M_j \in \mathsf{TPart}, i \neq j} M_i \times M_j$ *is a set of edges called "movement edges."*
- V_M *is the set of all machine vulnerabilities.*
- V_E *is the set of all edge vulnerabilities.*
- $v_M : M \rightarrow 2^{|V_M|}$ *is a function associating a set of vulnerabilities with each machine.*
- $v_E : E \rightarrow 2^{|V_E|}$ *is a function that associates with each movement edge* (m, m'), *a set of vulnerabilities that can allow an attacker's payload to move from machine* m *to* m'.

Thus, an AG is a graph where the set M of nodes is a set of machines. $v_M(m)$ is the set of all possible vulnerabilities that might be present on machine m. An edge $(m_1, m_2) \in E$ implies the existence of at least one vulnerability in m_2 that allows an attacker to move from m_1 to m_2. For each edge $(m_1, m_2) \in E$, the set of vulnerabilities allowing an attacker to move from m_1 to m_2 is $v_E(m_1, m_2)$. Without loss of generality, we assume that $V_E \cap V_M = \emptyset$. AGs also include the concept of a trust partition $\mathsf{TPart} = \{M_1, \ldots, M_k\}$ — all machines within any M_i trust other machines within M_i and hence, when one of them is compromised, then all the others are as well. Thus, if an attacker penetrates M_i, then he can use any vulnerability in any machine in M_i to move to and compromise another machine.

Example 5.5. Fig. 5.2 shows an AG where the white boxes are the trust components and the grey boxes are the machines. In this AG, $\mathsf{TPart} = \{M_1, M_2, M_3, M_4\}$, machines m_3, m_4 are in M_3, and $v_M(m_3) = \{v_3\}$. Each machine's associated set of vulnerabilities is shown below the machine — for instance, machine m_4 has two vulnerabilities, v_4, v_5. The movement edge from m_2 to m_3 in this figure says that vulnerabilities v_{11} and v_{12} can be used to move from machine m_2 to m_3.

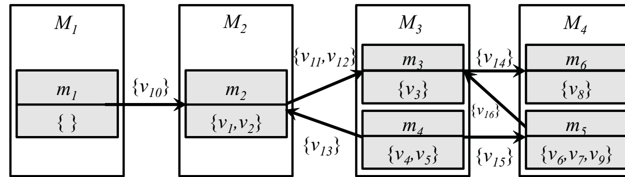


Fig. 5.2. Example of an attack graph

We can transform *any* AG into a VDG as follows. Given an AG $AG =< M, E, \mathsf{TPart}, V_M, V_E, v_M, v_E >$, let $AN = \{M \in \mathsf{TPart} \mid$ at least one $m \in M$ has in-degree $0\}$. Thus, AN is the *attacker network*, i.e. the set of trust components that can be directly attacked by an adversary. For instance, in

the AG of Example 5.5, we have $AN = \{M_1\}$. We can construct a VDG $G = (V', E')$ as follows.

Vertices of Transformation of an AG into a VDG

The set of vertices of the VDG is built as follows:

$$V' = \{(m, v) | m \in M, M \in \mathsf{TPart}, v \in v_M(m)\} \cup \\ \cup \{(m, v, m_1) | m \in M, M \in \mathsf{TPart} \setminus AN, (m_1, m) \in E, v \in v_E(m_1, m)\}.$$

In other words, the vulnerabilities in the VDG are constructed in two ways corresponding to the two terms in the union above:

- First, they include a set of pairs (m, v) where m is a machine and v is a vulnerability associated with that machine by AG;
- Second, they include a set of triples (m, v, m_1) where v is an edge vulnerability in the AG AG that allows an attacker to move from machine m to m_1.

Edges of Transformation of an AG into a VDG

Constructing the edges of the VDG associated with an AG AG is more complex. First we define two quantities. For each trust component $M \in \mathsf{TPart}$:

- Let $VN(M) = \{(m, v) | m \in M, v \in v_M(m)\}$ denote the set of all machine-vulnerability pairs;
- Let $AP(M) = \{(m, v, m_1) | m \in M, m_1 \in M, (m_1, m) \in E, v \in v_E((m_1, m))\}$ denote all triples (m, v, m_1) in which an attacker can leverage vulnerability v to move from machine m to m_1.

For each distinct pair of trust components $M_i, M_j \in \mathsf{TPart}$, we use $AP(M_i, M_j) = \{(m, v, m_1) | m_1 \in M_i, m \in M_j, (m_1, m) \in E, v \in v_E((m_1, m))\}$ to denote the set of all triples (m, v, m_1) where m is in the first trust component, m_1 is in the second, and v is a vulnerability that allows an attacker to move from m to m_1. In short, if $AP(M_i, M_j)$ is non-empty, then this means that an attacker who can compromise trust component M_i can also compromise trust component M_j.

We can now define the set E' of edges in the trust component as follows.

$$E' = \left(\bigcup_{M \in \mathsf{TPart} \setminus AN} AP(M) \times VN(M)\right) \cup \\ \cup \left(\bigcup_{M_i \in \mathsf{TPart}, M_j \in \mathsf{TPart} \setminus AN} AP(M_i) \times AP(M_i, M_j)\right).$$

Example 5.6. The above transformation converts the AG shown in Fig. 5.2 into the VDG shown in Fig. 5.3 below. For each trust component M, the white nodes represent the set $AP(M)$ while the grey nodes represent the set $VN(M)$. It is important to note that the VDG does not have any trust partition — the boxes denoting trust partitions from the AG for Fig. 5.1 are shown here merely to explain how each trust component in the AG got transformed into vertices of the associated VDG.

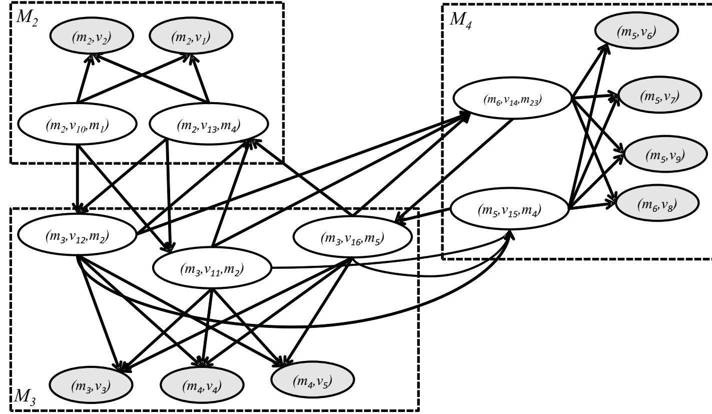


Fig. 5.3. Vulnerability dependency graph obtained from the attack graph in Fig. 5.2

Proposition 5.7. *Suppose nV_v and nV_e denote the maximum number of vulnerabilities within the nodes and labeling the edges of an* AG $AG =< M, E, \mathsf{TPart}, V_M, V_E, v_M, v_E >$*, respectively. Then, the* VDG *corresponding to AG can be computed with a polynomial-time algorithm running in* $O((|M| \cdot nV_v + |E| \cdot nV_e)^2)$ *time.*

Proof. We show a simple polynomial time algorithm that computes $VDG =< V', E' >$ from $AG =< M, E, \mathsf{TPart}, V_M, V_E, v_M, v_E >$. We assume we store TPart, v_M and v_E in hash structures — so the storage cost is less than or equal to $|M| \cdot nV_v + |E| \cdot nV_e$. It is easy to see that the time cost of computing all nodes in V is $O(|M| \cdot nV_v + |E| \cdot nV_e)$. Therefore, the time to enumerate all possible edge between the nodes in V is at most $O((|M| \cdot nV_v + |E| \cdot nV_e)^2)$. In the worst case, the cost of determining which of these edges is in E is the cost of enumerating all possible edges between the nodes in V. Verifying that an edge is in E can be done in constant time (since we use hash structures). Thus, the total cost of this algorithm is $O((|M| \cdot nV_v + |E| \cdot nV_e)^2)$.

5.4 Players' Strategy

In this section, we define the concept of a "strategy" for the defender and attacker.

5.4.1 Defender's Strategy

The enterprise security manager's task is to defend his organization from attacks. In order to do this, he needs to decide what software can run within his enterprise and what patches he should apply to them.

Definition 5.8 (Defender strategy). *A* defender strategy *is a pair* $\delta = (PR, PA)$ *where* $PR \subseteq \mathbf{PR}$ *and* $PA \subseteq \mathbf{PA}$*. We use* DS *to denote the set of all defender strategies.*

The definition above represents the fact that, even though **PR** may include several software products deployed on the enterprise network, the security officer decides to deactivate some of them, i.e. those in $(\mathbf{PR} \backslash PR)$. The security officer also decides to only install the set PA of patches.

In order to evaluate a defender strategy, we must understand the ramifications of the choices made by the strategy. Once we understand both the ramifications of different strategies, we can determine an optimal strategy.

Example 5.9. Consider the situation (products, vulnerabilities, and patches) shown in Example 5.2 and the cost and productivity values of Example 5.3. A possible defender strategy is $\delta_1 = (\{pr_1, pr_2, pr_3, pr_4\}, \{pa_4, pa_5\})$. In this strategy, the defender activates products pr_1, pr_2, pr_3, and pr_4, and applies patches pa_4 and pa_5. In this case we have $Tp(PR) = \sum_{i \in \{1,2,3,4\}} \mathsf{Prod}(pr_i) = 10$ and $Tc(PA) = \sum_{i \in \{4,5\}} \mathsf{CostD}(pa_i) = 9$. With strategy $\delta_2 = (PR', PA') = (\{pr_1, pr_2, pr_3, pr_4, pr_5\}, \{pa_1, pa_4\})$ we have $Tp(PR') = 15$ and $Tc(PA') = 5$.

Thus, any given defender strategy has an impact on productivity (capturing the dissatisfaction of those in the organization whose favorite products are deactivated) and cost (in terms of the patching cost). Of course, each strategy also has an implication in terms of how secure the enterprise is.

We now define what it means for a defender strategy to leave a vulnerability unprotected.

Definition 5.10 (Unprotected vulnerability). *Let* $G = (\mathbf{V}, E)$ *be a vulnerability dependency graph and* $\delta = (PR, PA) \in DS$ *be a defender strategy. A vulnerability* $v \in \mathbf{V}$ *is said to be* unprotected *iff all the following three conditions hold:*

- $v \in S(G)$ *or there exists a vulnerability* $v' \in in(G, v)$ *that is unprotected;*
- $PR(v) \cap PR \neq \emptyset$*;*
- $PA(v) \cap PA = \emptyset$*.*

Intuitively, for a strategy δ to leave a vulnerability unprotected, it must be unpatched (3rd condition above) and at least one product that is active must contain that vulnerability (2nd condition above). The first condition above says that if the vulnerability is a source vertex in the VDG, then we cannot protect against it as it is unpatched by the third condition. The second part of the first condition says that if there is an unprotected vertex v' that offers the capability to move from v' to v, then v is also unprotected. We use $UV(G, \delta)$ to denote the set of all unprotected vulnerabilities.

We now define the total vulnerability impact in the context of a selected defender strategy.

Definition 5.11 (Total vulnerability impact). *Suppose $\delta \in DS$ is a defender strategy, the total vulnerability impact* $\mathsf{tvi}(\delta)$ *of a defender strategy δ is given by*

$$\mathsf{tvi}(\delta) = \Sigma_{v \in UV(G,\delta)} \mathsf{Impact}(v).$$

Intuitively, this definition says that if v is a vulnerability, its impact is 0 if it is protected — otherwise its impact is whatever $\mathsf{Impact}(v)$ specifies (e.g. whatever the NVD says it is).

Example 5.12. Consider the two defender strategies δ_1 and δ_2 of Example 5.9.

- Under strategy δ_1, we have $UV(G,\delta_1) = \{v_1, v_2, v_4, v_5\}$ (the unprotected vulnerabilities under δ_1 are marked with a "*" in Fig. 5.1). In this case, the total impact of the unprotected vulnerabilities is $\mathsf{tvi}(\delta_1) = \sum_{v \in UV(G,\delta_1)} \mathsf{Impact}(v) = 22$.
- Under strategy δ_2, we have $UV(G,\delta_2) = \{v_2\}$ (marked with a "+" in Fig. 5.1). Observe that, although both v_5 and v_6 have an associated product that is activated and unpatched, they are protected since there is no path of unprotected vulnerabilities from v_1 or v_2 to them. In this case, $\mathsf{tvi}(\delta_2) = 4$.

We will discuss what constitutes an optimal defender strategy later as this depends upon what the attacker might do — the subject of the next subsection.

5.4.2 Attacker Strategy

Intuitively, an attacker's strategy is a set of vulnerabilities that he will use in order to penetrate the network. However, while the defender has full access to his enterprise's VDG, the attacker most likely does not. He must uncover vulnerabilities by using machines he has already compromised in order to discover what further vulnerabilities are present. We make the worst case assumption that the attacker can probe the network and, in the initial step, discover vulnerabilities in $S(G)$. Then, by exploiting these initial vulnerabilities, the attacker can probe the local network and discover other vulnerabilities (not in $S(G)$). This procedure continues in an iterative manner. My model is therefore a Stackelberg leadership model where the leader (the defender) moves before the follower (the attacker) moves [122].

Before formally defining an attacker strategy, we recall that if $G = (V, E)$ is a graph and $V' \subseteq V$ is a set of vertices, then the subgraph induced by V' is the graph $G' = (V', \{(a,b) | a, b \in V', (a,b) \in E\})$. In other words, the subgraph induced by V' has V' as the set of vertices and retains all those edges in E whose end-points are both in V'.

Definition 5.13 ((Valid) Attacker strategy). *An* attacker strategy *is a set of vulnerabilities $\alpha \subseteq \mathbf{V}$ such that*

1. $\alpha \cap S(G) \neq \emptyset$ *and*
2. for each vertex $v \in \alpha$*, there exists a path from a vertex in* $\alpha \cap S(G)$ *to* v *in the subgraph of* G *induced by* α*.*

An attacker strategy α *is* valid *w.r.t. a defender strategy* δ *iff for all vulnerabilities* $v \in \alpha$*, it is the case that* $v \in UV(G, \delta)$*.*

We use AS to denote the set of all attacker strategies and AS_δ to denote the set of all attacker strategies that are valid w.r.t. a defender strategy δ.

Simply put, an attacker strategy is a set of vertices such that each vertex is reachable from a vertex in $S(G)$. It is important to note that an attacker strategy is not guaranteed to work because the defender might have protected it. The impact of a valid attacker strategy w.r.t. a defender strategy is given below.

Definition 5.14 (Impact of a valid attacker strategy w.r.t. a defender strategy). *Let* δ *be a defender strategy, and* α *be an attacker strategy that is valid w.r.t.* δ*. We define the* impact of α *in two ways:*

$$\mathsf{impactDA1}(\alpha) = \max_{v \in \alpha} \mathsf{Impact}(v)$$

$$\mathsf{impactDA2}(\alpha) = \sum_{v \in \alpha} \mathsf{Impact}(v)$$

impactDA1 captures the idea that the impact generated by an attack is the maximum of the impacts of the different vulnerabilities that are exploited by the attacker. impactDA2 says the impact of an attack is the sum of these impacts. In the chapter, we use the notation impactDA to represent either impactDA1 or impactDA2.

The following example shows this situation.

Example 5.15. Consider again the two defender strategies δ_1 and δ_2 of Example 5.9.

- Under strategy δ_1, valid attacker strategies are $\alpha_1 = \{v_1\}$, $\alpha_2 = \{v_1, v_4\}$, $\alpha_3 = \{v_1, v_4, v_5\}$, and $\alpha_4 = \{v_2\}$. We have
  - $\mathsf{impactDA1}(\alpha_1) = \mathsf{impactDA1}(\alpha_2) = \mathsf{impactDA1}(\alpha_3) = 7$
  - $\mathsf{impactDA1}(\alpha_4) = \mathsf{impactDA2}(\alpha_4) = 4$
  - $\mathsf{impactDA2}(\alpha_1) = 7$, $\mathsf{impactDA2}(\alpha_2) = 12$
  - $\mathsf{impactDA2}(\alpha_3) = 18$
- Under strategy δ_2, the only valid attacker strategy is $\alpha = \{v_2\}$ so we have $\mathsf{impactDA1}(\alpha) = \mathsf{impactDA2}(\alpha) = 4$.

We assume that the attacker has a utility function $\mathsf{utilA} : \mathbf{V} \to \mathbb{R}_0^+$ associating a utility (to him) of exploiting a vulnerability, as well as a cost function $\mathsf{costA} : \mathbf{V} \to \mathbb{R}_0^+$ associating the cost (to him) of exploiting a vulnerability. For instance, utilA can be derived from NIST's Common Vulnerability Scoring System (e.g. via their impact metric) and the cost can, for example, be derived from CVSS's access complexity metric that measures the difficulty for a vulnerability to be exploited by an adversary.

Definition 5.16 (Preferred attacker strategies). *Given a utility threshold u and a cost threshold c, the set of preferred attacker strategies is defined as*

$$PAS(c,u) = \{\alpha | \alpha \in AS, \sum_{v\in\alpha} \mathsf{utilA}(v) \geq u, \sum_{v\in\alpha} \mathsf{costA}(v) \leq c\}.$$

The set of preferred valid attacker strategies w.r.t. a defender strategy δ is defined as

$$PAS_\delta(c,u) = \{\alpha | \alpha \in AS_\delta, \sum_{v\in\alpha} \mathsf{utilA}(v) \geq u, \sum_{v\in\alpha} \mathsf{costA}(v) \leq c\}$$

Thus, the preferred attacker strategies are the strategies with utility exceeding a threshold value and cost below a threshold value. Cost may not be a dollar value — it could be a measure of the risk of being discovered (e.g. if the attacker is a nation state, the risk of discovery may have political or kinetic consequences) or it could be a measure of the time and effort required to launch such attacks. Thus, the attacker wants to do the best he can under his operating constraints. The following result characterizes the maximal value of impact (under both definitions $\mathsf{impactDA1}, \mathsf{impactDA2}$).

Proposition 5.17 *Suppose δ is a defender strategy. If $c = \infty$, then the maximum impact value is*

$$\max_{\alpha \in PAS_\delta(\infty,u)} \mathsf{impactDA}(\alpha) = \begin{cases} \mathsf{impactDA}(UV(G,\delta)) & \text{if } \sum_{v\in UV(G,\delta)} \mathsf{utilA}(v) \geq u \\ 0 & \text{otherwise.} \end{cases}$$

Proof. $UV(G,\delta)$ is a trivially valid attacker strategy. When $c = \infty$, the constraint $\sum_{v\in\alpha} \mathsf{costA}(v) \leq c$ can be deleted. By Definition 5.14 we know that each strategy $\alpha \in AS$ containing vulnerabilities in $\mathbf{V} \setminus UV(G,\delta)$ is not a valid strategy. By Definition 5.14, any subset $\alpha' \subseteq UV(G,\delta)$ has an impact of $UV(G,\delta)$ at most: $\sum_{v\in\alpha'} \mathsf{utilA}(v) \leq \sum_{v\in UV(G,\delta)} \mathsf{utilA}(v)$. The result follows.

This result provides an upper bound on the impact of the attacker's preferred attack strategy by considering the case when the attacker has no cost constraint. In reality, the attacker has cost constraints to consider, and so he may not be able to launch all possible attacks.

The following result says that checking whether there exists an attacker strategy satisfying the attacker's cost and utility constraints is intractable — this is good news for defenders as the attacker's problem is computationally hard.

Theorem 5.18 *Checking if there exists $\alpha \in PAS(c,u)$ is* **NP***-complete (for both* $\mathsf{impactDA1}$ *and* $\mathsf{impactDA1}$*)*

Proof. Membership in **NP** follows from the fact that we can guess, in nondeterministic polynomial time, a set of vulnerabilities $\alpha \in \mathbf{V}$, and then check in deterministic polynomial time if $\alpha \in PAS(c,u)$. In fact, the check consists of verifying all the following four conditions:

1. $\alpha \cap S(G) \neq \emptyset$;
2. for each vertex $v \in \alpha$, there exists a path from a vertex in $\alpha \cap S(G)$ to v in the subgraph of G induced by α (this can be checked using a polynomial time reachability algorithm);
3. $\sum_{v \in \alpha} \mathsf{utilA}(v) \geq u$;
4. $\sum_{v \in \alpha} \mathsf{costA}(v) \leq c$.

We prove **NP**-hardness by reduction from SAT, which is known to be **NP**-complete [133]. Given a boolean formula ϕ in conjunctive normal form (CNF), the SAT problem is to decide if there exists a truth assignment to the variables in ϕ that satisfies ϕ. Given a CNF boolean formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_l$ where each clause C_i is a disjunction of propositional variables (negated or not), let $X = \{x_1, \ldots, x_k\}$ be the set of propositional variables in ϕ. We can built a vulnerability dependency graph $G = (V, E)$ where

$$V = \{x_1, \overline{x}_1, \ldots, x_k, \overline{x}_k\} \cup \{C_1, \ldots, C_l\} \cup \{f\}$$

$$E = \{(a, b) | a \in \{x_i, \overline{x}_i\}, b \in \{x_{i+1}, \overline{x}_{i+1}\}, i \in \{1, \ldots, k-1\}\} \cup \{(x_k, f), (\overline{x}_k, f)\} \cup \\ \cup \{(a, C_j) | a \in \{x_i, \overline{x}_i\}, i \in \{1, \ldots, k\}, j \in \{1, \ldots, l\}, a \text{ appears in } C_j\}$$

According to this construction, we have two vulnerabilities x_i and $\overline{x}_i$ for each variable $x_i \in X$, representing whether x_i is $true$ or $false$, respectively. Moreover, we have a vulnerability for each clause C_i plus another vulnerability f indicating if the truth assignment is feasible, i.e. only one truth value is assigned to each variable. The resulting graph is shown in Fig. 5.4, where the edges from vertices $\{x_1, \overline{x}_1, \ldots, x_k, \overline{x}_k\}$ to vertices $\{C_1, \ldots, C_l\}$ are not specified because they depend on the specific CNF formula.

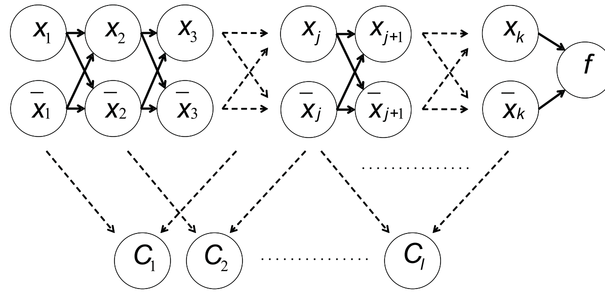


Fig. 5.4. From CNF formula to vulnerability dependency graph

In addition, we have that $S(G) = \{x_1, \overline{x}_1\}$, because each vertex C_i has at least one edge, since each clause has at least one variable.

We define the utility and cost functions for the attacker in the following way:

$$\mathsf{utilA}(a) = \begin{cases} 1 & \text{if } a = f \\ 1 & \text{if } a \in \{C_1, \dots, C_l\} \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{costA}(a) = \begin{cases} 1 & \text{if } a \in \{x_1, \dots, x_k\} \\ 1 & \text{if } a \in \{\overline{x}_1, \dots, \overline{x}_k\} \\ 0 & \text{otherwise} \end{cases}$$

and set the minimum utility $u = l + 1$ and the maximum cost $c = k$. Since we need to satisfy the constraint $\sum_{v \in \alpha} \mathsf{utilA}(v) \geq l + 1$, each attacker strategy $\alpha \in PAS(c, u)$ must contain the vulnerabilities $\{C_1, C_1, \dots, C_l, f\}$ because they are the only vulnerabilities with utility 1, while all the others have utility 0. However, since the vertices $\{C_1, C_1, \dots, C_l, f\}$ are not in $S(G)$, then in order to satisfy the constraint that each vulnerability in α must be reachable from a vertex in $S(G)$, each α must contain other vulnerabilities in $\{x_1, \overline{x}_1, \dots, x_k, \overline{x}_k\}$. As we can see in Fig. 5.4, the only way to reach f from $S(G) = \{x_1, \overline{x}_1\}$, is that each α contains x_i or $\overline{x}_i$ for each $i \in 1, \dots, k$. Moreover, because of the constraint $\sum_{v \in \alpha} \mathsf{costA}(v) \leq c$, we have that α does not contain both x_i or $\overline{x}_i$ for each $i \in 1, \dots, k$. Therefore, in order to reach f we have that each attacker strategy α contains a feasible truth assignment for the variables in X, and then to reach all the vulnerabilities $\{C_1, C_1, \dots, C_l\}$ from $S(G)$ we also need the feasible truth assignment to satisfy ϕ. Finally, by using this transformation, we have that all the strategies in $PAS(c, u)$ represent all feasible truth assignment of the variables X that satisfy ϕ. Therefore, hardness of the problem holds. Note that the result holds for both impactDA1 and impactDA1.

At this point, since the number of attacker strategies in $PAS(c, u)$ can be exponential, the question is whether it is possible to enumerate all attacker strategies using a polynomial total time algorithm. We recall that an algorithm generating all configurations that satisfy a given specification is said to be *polynomial total time* [134] if the time required to output all configurations is bounded by a polynomial in n (the size of the input) and C (the number of configurations). Unfortunately, as stated in Proposition 5.19, this is not possible unless $\mathbf{P} = \mathbf{NP}$.

Proposition 5.19 *If there exists a polynomial total time algorithm for generating all the attacker strategies in $PAS(c, u)$, then $\mathbf{P} = \mathbf{NP}$.*

Proof. The proof is inspired by [135]. Suppose there exists an algorithm $\mathcal{A}$ running in time $\varphi(n, C)$ that generates C different attacker strategies in $PAS(c, u)$, where φ is a polynomial function of n and C for an instance of size n. We can then execute $\mathcal{A}$ and stop it after a time equal to $\varphi(n, 1)$. If $\mathcal{A}$ does not terminate or terminates outputting one attacker strategy in $PAS(c, u)$, then there exists at least one $\alpha \in PAS(c, u)$. Otherwise, does not exist $\alpha \in PAS(c, u)$. We thus have a polynomial algorithm to solve an **NP**-hard problem (Theorem 5.18). This is impossible unless **P**=**NP**.

Corollary 5.20. *Given a defender strategy δ, Theorem 5.18 and Proposition 5.19 also hold for $PAS_\delta(c, u)$.*

5.4.3 Best Strategy of the Attacker

The success of an attacker strategy depends on the strategy of the defender. In this case, let δ^+ be a defender strategy and c the threshold cost. The best attacker strategy of the attacker is defined as follows:

$$\alpha^* = arg\,max_{\alpha \in AS_{\delta^+}, (\sum_{v \in \alpha} \mathsf{costA}(v) \leq c)} \sum_{v \in \alpha} \mathsf{utilA}(v)$$

The best attacker strategy can be computed by the integer linear program (ILP) given below. We have for each $v_i \in UV(G, \delta^+)$ a binary variable $k_i \in \{0, 1\}$ that is set to 1 if the attacker chooses to use vulnerability v in its strategy and 0 otherwise. Then the ILP is:

$$\textbf{maximize} \quad \sum_{v_i \in UV(G,\delta^+)} \mathsf{utilA}(v) \cdot k_i \tag{5.1}$$

subject to

$$\sum_{v_i \in UV(G,\delta^+)} \mathsf{costA}(v_i) \cdot k_i \leq c \tag{5.2}$$

$$k_i - \sum_{v_j \in in(G,v_i)} k_j \leq 0 \qquad v_i \in UV(G,\delta^+) \setminus S(G) \tag{5.3}$$

$$k_i \in \{0,1\} \qquad v_i \in UV(G,\delta^+) \tag{5.4}$$

where the objective function represents the utility produced by the attacker strategy described by variables k, and Constraint 5.2 models the fact that the total cost of the attacker strategy must be less than or equal to a threshold value c. Constraints 5.3 model the fact that the attacker can use a vulnerability $v_i \notin S(G)$ if he exploits at least one vulnerability in $in(G, v_i)$. Corollary 5.20 allows us to infer the following result.

Corollary 5.21. *Given a defender strategy δ, finding the best attacker strategy is* **NP***-hard.*

5.5 Pareto Analysis for the Defender

Each attacker strategy in $PAS_\delta(c, u)$ has an actual impact w.r.t. each defender strategy $\delta \in DS$. This impact is measured via either $\mathsf{impactDA1}$ and $\mathsf{impactDA2}$. The defender wants to find a strategy that minimizes the maximum impact produced by the attacker. The defender can select this best strategy by either deactivating software products running on his enterprise or by installing additional patches. One extreme solution is to deactivate all software and patch nothing — this preserves security at the price of not allowing anyone to do their jobs.

5.5.1 Bi-Objective Optimization Problem Formulation

We formulate the best defender strategy as a bi-objective optimization problem. We assume that there is a maximal cost c the attacker is willing to bear and we also assume the attacker requires that his attacks have a minimal utility. The cost could, for example, be measured using CVSS's access complexity metric, while the utility could be measured by CVSS's impact metric. Likewise, the defender has a maximum cost $\hat{c}$ he is willing to bear for applying the selected patches (e.g. time taken or dollar cost of the labor involved). Thus, the bi-objective optimization problem for the defender is:

$$\min_{\delta=(PR,PA)\in DS, Tc(PA)\leq\hat{c}} \{ \max_{\alpha\in PAS_{\delta}(c,u)} \mathsf{impactDA}(\alpha), -Tp(PR)\} \quad (5.5)$$

The first part ($\max_{\alpha\in PAS_{\delta}(c,u)} \mathsf{impactDA}(\alpha)$) of this bi-objective optimization problem tries to minimize the maximum impact produced by the attacker — the second part simultaneously tries to minimize the negative of total productivity ($-Tp(PR)$) which is equivalent to maximizing the total productivity, subject to the cost constraint.

Brief Overview of Pareto Optimization. These two objective functions compete — in fact, we can minimize impact by eliminating productivity altogether and any increase in productivity increases the impact of the attacker as well. Pareto analysis [136] is a classic method used to carry out optimizations in situations where there are multiple competing objectives that must somehow be satisfied simultaneously. The basic idea behind Pareto optimization of two competing objective functions ϕ_1 and ϕ_2 subject to a set $\mathcal{C}$ of constraints is as follows. Suppose σ, σ' are two different solutions and suppose that both ϕ_1 and ϕ_2 are maximization problems.[3] We say that σ is worse than σ', denoted $\sigma \lhd \sigma'$ iff (v_1, v_2) is the pair of values assigned by σ to the objective function ϕ_1 and (v_1', v_2') is the pair of values assigned by σ' to the objective function ϕ_2 and $v_1 \leq v_1'$ and $v_2 \leq v_2'$. In this case, σ' gives better values to both objective functions and hence is intuitively better. A solution σ is said to be *Pareto optimal* w.r.t. maximization problems ϕ_1, ϕ_2 and constraints $\mathcal{C}$ if and only there is no solution $\sigma' \neq \sigma$ such that $\sigma \lhd \sigma'$.

We use Pareto analysis to solve the optimization problem (5.5) in order to find a compromise strategy. In our case, the competing functions are the maximum impact allowed and the productivity. The main point of this method is the computation of the Pareto frontier (see [136]).

The following result says that checking for existence of a defender strategy whose cost is less than $\hat{c}$, whose productivity is at least p, and where the impact of attacks that can pierce the strategy is at most r is intractable at the $\boldsymbol{\Sigma_2^P}$-completeness level under the assumption that the attacker will only select a strategy with cost less than or equal to c and utility u (to him).

[3] There is no loss of generality in this as minimizing x is the same as maximizing $-x$.

Theorem 5.22 *Given the real numbers* p, r, $\hat{c}$, c *and* u, *the problem of deciding whether there exists a defender strategy* $\delta = (PR, PA) \in DS$ *such that* $Tc(PA) \leq \hat{c}$, $\max_{\alpha \in PAS_{\delta}(c,u)} \mathsf{impactDA}(\alpha) \leq r$, *and* $Tp(PR) \geq p$ *is* $\boldsymbol{\Sigma_2^P}$*-complete (for both* impactDA1 *and* impactDA2*).*

Proof. Membership in $\boldsymbol{\Sigma_2^P}$ follows from the fact that we can guess, in nondeterministic polynomial time, a defender strategy (PR, PA) and check if all the following three condition are satisfied.

1. $Tc(PA) \leq \hat{c}$;
2. $Tp(PR) \geq p$;
3. $\max_{\alpha \in PAS_{(PR,PA)}(c,u)} \mathsf{impactDA}(\alpha) \leq r$.

Conditions (1) and (2) can be verified in deterministic polynomial time, while the third condition needs an **NP** oracle verifying that there does not exist an attacker strategy $\alpha \in PAS_{(PR,PA)}(c,u)$ such that $\mathsf{impactDA}(\alpha) > r$. This oracle guesses (in nondeterministic polynomial time) an attacker strategy $\alpha \in PAS(c,u)$ (by using the check phase done in the proof of Theorem 5.18) and checks (in deterministic polynomial time) if α is valid w.r.t. δ and $\mathsf{impactDA}(\alpha) > r$.

We prove $\boldsymbol{\Sigma_2^P}$-hardness by reduction from $\exists\forall$ SAT, that is known to be Σ_2^P-complete if the boolean formula considered is in disjunctive normal form (DNF) [133]. Given a DNF boolean formula ϕ, let $Y = \{y_1, \ldots, y_h\}$ and $X = \{x_1, \ldots, x_k\}$ be two sets of variables in ϕ. The $\exists\forall$ SAT problem consists in deciding whether there exists an assignment of Y variables such that for all assignment of X variables, ϕ is satisfied. We recall that a DNF boolean formula $C_1 \vee C_2 \vee \cdots \vee C_l$ is a disjunction of clauses where each clause C_i' is a conjunction of variables (negated or not). The reduction from $\exists\forall$ SAT to our problem is the following. Given a DNF boolean formula ϕ with variables $X = \{x_1, \ldots, x_k\}$, we build a vulnerability dependency graph $G = (V, E)$ where

$$
\begin{aligned}
V = \; & \{ya_1, \ldots, ya_h\} \cup \{y_1, \overline{y}_1, \ldots, y_h, \overline{y}_h\} \cup \{x_1, \overline{x}_1, \ldots, x_k, \overline{x}_k\} \cup \\
& \cup \{C_1, \ldots, C_l\} \cup \{f\}
\end{aligned}
$$

$$
\begin{aligned}
E = \; & \{(a,b) | a \in \{x_i, \overline{x}_i\}, b \in \{x_{i+1}, \overline{x}_{i+1}\}, i \in \{1, \ldots, k-1\}\} \cup \\
& \cup \{(x_k, f), (\overline{x}_k, f)\} \cup \\
& \cup \{(a, C_j) | a \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}, i \in \{1, \ldots, k\}, j \in \{1, \ldots, l\}, -a \in C_j\}
\end{aligned}
$$

According to the above definition we have, for each variable in the DNF $y_i \in Y$ (resp. $x_j \in X$), two vulnerabilities, i.e. y_i and $\overline{y_i}$ (resp. x_j and $\overline{x_j}$), representing the truth values of each variable. Moreover, we have, for each $y_i \in Y$, a variable ya_i whose aim is to guarantee a valid truth assignment for y_i. In addition, for each clause C_z we have the vulnerability C_z indicating that, if C_z is present in the attacker strategy, the corresponding clause C_z in the formula is not satisfied. The edges are reported in Fig. 5.5, where the vulnerabilities are represented as circles and the patches as squares. An

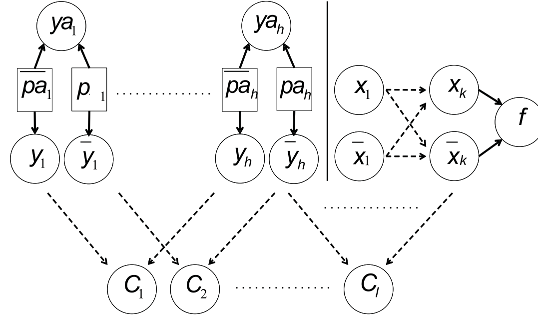


Fig. 5.5. From DNF formula to vulnerabilities dependency graph in $\exists\forall SAT$ reduction

edge between two vulnerabilities represents an edge in the graph G, while an edge between a patch and a vulnerability means that the patch solves the vulnerability. The edges among the vulnerabilities in $\{x_1, \overline{x}_1, \ldots, x_k, \overline{x}_k\}$ are the same of the proof of Theorem 5.18, whereas there is an edge between the vulnerabilities y_i (resp. x_j) and a vulnerability C_z if the variable y_i (resp. x_j) appears in the clause C, and there is an edge between the vulnerabilities $\overline{y_i}$ (resp. $\overline{x_j}$) and a vulnerability C_z if the variable y_i (resp. x_j) appears negated in the clause C_z.

We considered the set of patches $PA = \{pa_1, \overline{pa}_1, \ldots, pa_h, \overline{pa}_h\}$ where each patch pa_i (resp. $\overline{pa}_i$) solves both the vulnerabilities $\overline{y}_i$ (resp. y_i) and ya_i. We define the utility and cost functions for the attacker as follows:

$$\mathsf{utilA}(a) = \begin{cases} 1 & if \;\; a = f \\ 1 & \text{if } a \in \{C_1, \ldots, C_l\} \\ 1 & \text{if } a \in \{y_1, \overline{y}_1, \ldots, y_h, \overline{y}_h\} \\ 0 & \text{otherwise} \end{cases} \quad \mathsf{costA}(a) = \begin{cases} 1 & \text{if } a \in \{x_1, \ldots, x_k\} \\ 1 & \text{if } a \in \{\overline{x}_1, \ldots, \overline{x}_k\} \\ 0 & \text{otherwise} \end{cases}$$

and we impose that the minimum utility is $u = l + h + 1$ and the maximum cost is $c = k$.

For the defender we fix the cost of every patch to 1, and the impact of a vulnerability as

$$\mathsf{Impact}(a) = \begin{cases} 1 & \text{if } a = f \\ 1 & \text{if } a \in \{ya_1, \ldots, ya_h\} \\ 0 & \text{otherwise} \end{cases}$$

We impose that the maximum cost budget is $\hat{c} = h$ and the maximum impact is $r = 0$. Moreover, we assume that p is equal to the sum of all productivities of each product, i.e., in this situation, no product can be disabled.

From this reduction, we see that the defender can only decide which patches to use and, since $\hat{c} = h$, he can use h patches at most. Moreover, since the r must be 0, each vulnerability ya_i must be patched, and

therefore, for each $1 \leq i \leq h$, the defender will use pa_i or $\overline{pa}_i$. It follows that the patches used by the defender represent the truth assignment for the variables y_i. Note that the source vertices of the graph G are $S(G) = \{y_1, \overline{y}_1, \ldots, y_h, \overline{y}_h\} \cup \{ya_1, \ldots, ya_h\} \cup \{x_1, \overline{x}_1, \ldots, x_h, \overline{x}_k\}$. On the attacker side, instead, we have that, because of the attacker utility constraints, the set of vulnerabilities used by the attacker must include: (1) all the vulnerabilities C_z, (2) the vulnerability f, and (3) h vulnerabilities in $\{y_1, \overline{y}_1, \ldots, y_h, \overline{y}_h\}$. It follows that if the defender uses patch pa_i, then the attacker will use the vulnerability y_i, whereas if the defender uses patch $\overline{pa}_i$, then the attacker will use the vulnerability $\overline{y}_i$. Thus, the vulnerabilities $y_i, \overline{y}_i$ represent a valid truth assignment for the variables y_i chosen by the defender by using the patches. Moreover, as shown in the proof of Theorem 5.18, because of the cost constraints, and since f must be present in the attacker strategy, only one between x_j and $\overline{x}_j$ can be present in the attacker strategy, then also the vulnerabilities $x_j, \overline{x}_j$ represent a valid truth assignment for the variables x_j. It follows that, since all the vulnerabilities C_z must be present in the attacker strategy, the only possibility is to have an assignment of the variables X and Y that does not satisfy the DNF. Finally, since $r = 0$, the defender will have to find a truth assignment for the variables Y such that there does not exist a truth assignment of the variables X which causes the DNF to be unsatisfied — this corresponds to the $\exists\forall$ SAT problem. This proves Σ_2^P-hardness of our problem. Note that, as $r = 0$, the result holds for both impactDA1 and impactDA2.

The theorem below considers the same case as the theorem above with one change — this time, the defender makes no assumption at all about the cost the attacker is willing to bear (i.e. that cost is ∞). This represents a kind of worst case scenario from the defender's perspective. Under this assumption, the problem of checking if a defender strategy exists is still intractable, but at the **NP**-complete level.

Theorem 5.23 *Given the real numbers $p, r, \hat{c}, u$, if $c = \infty$ then the problem of deciding whether there exists a defender strategy $\delta = (PR, PA) \in DS$ such that $Tc(PA) \leq \hat{c}$, $\max_{\alpha \in PAS_\delta(\infty,u)} \mathsf{impactDA}(\alpha) \leq r$, and $Tp(PR) \geq p$ is* **NP***-complete (for both* impactDA1 *and* impactDA2*).*

Proof. Membership in **NP** follows from the fact that we can guess, in nondeterministic polynomial time, a defender strategy (PR, PA) and then check if all the following three conditions are satisfied.

1. $Tc(PA) \leq \hat{c}$;
2. $Tp(PR) \geq p$;
3. $\max_{\alpha \in PAS_{(PR,PA)}(\infty,u)} \mathsf{impactDA}(\alpha) \leq r$.

Conditions (1) and (2) can be verified in deterministic polynomial time, and, based on Proposition 5.17, Condition (3) can be verified in polynomial time as well.

We prove **NP**-hardness by reduction from SET COVER, which is known to be **NP**-complete [133]. The SET COVER problem is the following: given a set $S = \{s_1, \ldots, s_m\}$ and a family $\mathcal{F} = \{S_1, \ldots, S_s\}$ of subsets of S, decide whether there exists a subset C of $\mathcal{F}$ such that $|C| \leq k$ and $\bigcup_{S_i \in C} S_i = S$. The reduction is the following. For each element $s \in S$ we have a vulnerability v_s in the dependency graph G, and this dependency graph has no edges, thus, all the vulnerabilities are in $S(G)$. For each $S_i \in \mathcal{F}$ we have a patch p_{S_i} that solves the set of vulnerabilities $\{v_s | s \in S_i\}$. Moreover, each vulnerability in the graph has impact 1, each patch has cost 1, and the value of u is 0. We also set $\hat{c} = k$ and $r = 0$. We assume that p is equal to the sum of the productivities of each product, i.e., no product is deactivated. Thus, in this reduction, our problem consists of verifying if there exists a set of k patches or less that solve each vulnerability (since $r = 0$). This is equivalent to the SET COVER problem, and hence our statement follows. Note that as $r = 0$, the result holds for both impactDA1 and impactDA2.

5.5.2 Computing the Pareto Frontier

In order to perform our Pareto analysis, let

$$VP = \{(\max_{\alpha \in PAS_\delta(c,u)} \mathsf{impactDA}(\alpha), -Tp(PR)) \mid \delta = (PR, PA) \in DS,\ Tc(PA) \leq \hat{c}\}$$

be the set of all possible pairs of values for our two objectives, given that cost constraints are satisfied. Given two pairs $(a, b), (a', b') \in VP$, we say that (a, b) *dominates* (a', b') if $(a \leq a' \wedge b < b') \vee (a < a' \wedge b \leq b')$.

Definition 5.24 (Pareto frontier). *The* Pareto frontier PF *for the bi-objective optimization problem (5.5) is the set* $\{(a, b) \mid (a, b) \in VP$ *and* $\nexists (a', b') \in VP$ *such that* (a', b') *dominates* $(a, b)\}$.

Algorithm $FindPF$ (Algorithm 1) computes the Pareto Frontier of the the bi-objective optimization problem (5.5) without enumerating all pairs in VP. The algorithm uses two functions $getp(p^+, \hat{c}, c, u)$ and $getr(r^+, \hat{c}, c, u)$.

When the optimization problem associated with $getp(r^+, \hat{c}, c, u)$ has no feasible solution, the function $pget(r^+, \hat{c}, c, u)$ returns $Null$. Note that this multi-objective optimization problem is not linear — later in Section 5.5.4, We will show how it can be represented as an integer linear optimization problem.

The following result looks at the complexity of computing the Pareto Frontier.

Theorem 5.25 *Given the real numbers c, $\hat{c}$ and u, the problem of checking whether a point (p^+, r^+) is in the Pareto Frontier is in $\boldsymbol{\Sigma_2^P} \cap \boldsymbol{\Pi_2^P}$ and $\boldsymbol{\Sigma_2^P}$-hard (for both* impactDA1 *and* impactDA2*). Moreover, if $c = \infty$ the problem is in* **DP**=**NP** $\cap$ *co*-**NP** *and* **NP**-*hard (for both* impactDA1 *and* impactDA2*).*

Algorithm 1 $FindPF$

```
1: procedure FindPF(ĉ, c, u)
2:     PF = ∅;
3:     p⁺ = getp(∞, ĉ, c, u);
4:     while (p⁺ ≠ Null) do
5:         r⁺ = getr(p⁺, ĉ, c, u);
6:         PF = PF ∪ {(p⁺, r⁺)}
7:         p⁺ = getp(−r⁺, ĉ, c, u);
8:     end while
9:     return PF;
10: end procedure
```

where:

$$getp(r^+, \hat{c}, c, u) = \min_{\substack{(PR,\,PA) \in DS \\ Tc(PA) \leq \hat{c} \\ \max_{\alpha \in PAS_{(PR,PA)}(c,u)} \mathsf{impactDA}(\alpha) < r^+}} -Tp(PR)$$

$$getr(p^+, \hat{c}, c, u) = \min_{\substack{(PR,\,PA) \in DS \\ Tc(PA) \leq \hat{c} \\ Tp(PR) = p^+}} \max_{\alpha \in PAS_{(PR,PA)}(c,u)} \mathsf{impactDA}(\alpha)$$

Proof. The problem of checking whether a point (p^+, r^+) is in the Pareto Frontier can be solved by verifying the following two statements:

1. there exists a defender strategy $\delta = (PR, PA) \in DS$ such that
   - $Tc(PA) \leq \hat{c}$
   - $\max_{\alpha \in PAS_\delta(c,u)} \mathsf{impactDA}(\alpha) \leq r^+$
   - $Tp(PR) \geq p^+$
2. does not exist a defender strategy such that $Tc(PA) \leq \hat{c}$, and either
   - $\max_{\alpha \in PAS_\delta(c,u)} \mathsf{impactDA}(\alpha) < r^+$, and $Tp(PR) \geq p^+$

   or
   - $\max_{\alpha \in PAS_\delta(c,u)} \mathsf{impactDA}(\alpha) \leq r^+$, and $Tp(PR) > p^+$.

It is easy to see from the proof of Theorem 5.22 that the problem (1) is in $\boldsymbol{\Sigma_2^P}$, while the problem (2) in $\boldsymbol{\Pi_2^P}$. The hardness results directly follows from Theorem 5.22 the fact that the first problem is in $\boldsymbol{\Sigma_2^P}$. In particular, if $c = \infty$, from the proof of Theorem 5.23, we have that the problem (1) is in **NP**, while the problem (2) is in co-**NP**. The **NP**-hardness derives from the proof of Theorem 5.23, too.

The following result looks at the complexity of finding the Pareto Frontier in the case of impactDA1 when the defender makes the worst-case assumption that the attacker has no cost constraints at all to consider (e.g. if the attacker is a nation state that may be willing to bear huge costs).

Proposition 5.26 *Computing the Pareto Frontier is in* $\mathbf{FP}^{\mathbf{\Sigma_2^P}}$ *by considering* impactDA1. *Moreover if* $c = \infty$ *the problem is in* $\mathbf{FP}^{\mathbf{NP}}$.

Proof. Since we are considering impactDA1, it is possible to enumerate all the values that r can assume in polynomial time. In order to prove the proposition we provide an algorithm in $\boldsymbol{FP}^{\boldsymbol{\Sigma_2^P}}$ that is the following:

1. compute the set IM of all impact values for each vulnerability (linear time in the size of the vulnerability dependency graph);
2. for each value $r \in IM$, compute the value of the maximum productivity p by using a binary search algorithm with an oracle deciding whether there exists a defender strategy $\delta = (PR, PA) \in DS$ such that $Tc(PA) \leq \hat{c}$, and $\max_{\alpha \in PAS_\delta(c,u)} \mathsf{impactDA}(\alpha) \leq r$, and $Tp(PR) \geq p$ (in $\boldsymbol{FP}^{\boldsymbol{\Sigma_2^P}}$ from Theorem 5.22);
3. given the set of all pairs (r, p), obtained from steps 1 and 2, compute all the non-dominated points in this set (in polynomial time since the size of IM is polynomial in the size of G).

It follows that the problem of computing the Pareto Frontier by considering impactDA1 is in $\boldsymbol{FP}^{\boldsymbol{\Sigma_2^P}}$. When $c = \infty$, from the proof of Theorem 5.23, the step (2) can be done in $\mathbf{FP}^{\mathbf{NP}}$.

It is worth noting that the same proof done for Proposition 5.26 cannot be valid for the case considering impactDA2 as the set of all possible values that r can assume is not enumerable in polynomial time.

5.5.3 Finding the Optimal Defender Strategy

Once we compute the Pareto frontier, we are able to choose a compromise between impact and productivity, i.e., a Pareto point (r^+, p^+) in the Pareto frontier. Once a Pareto point is chosen, we can compute an *Optimal Defender Strategy* using the productivity generated by the Pareto point to minimize the maximal impact produced by the attacker. This is formalized via the following program.

$$(PR, PA)^* = arg\,min_{\substack{(PR,PA) \in DS \\ Tc(PA) \leq \hat{c} \\ Tp(PR) = p^+ \\ \max_{\alpha \in PAS_{(PR,PA)}(c,u)} \mathsf{impactDA}(\alpha) \leq r^+}} \mathsf{tvi}((PR, PA))$$

The next corollary follows immediately from the proofs of Theorem 5.22 and Theorem 5.23.

Corollary 5.27. *Given the real numbers c, u, $\hat{c}$, p^+, r^+ and t, the problem of checking whether there exists a defender strategy (PR, PA) such that $Tc(PA) \leq \hat{c}$, ,$Tp(PR) = p^+$,$\max_{\alpha \in PAS_{(PR,PA)}(c,u)} \mathsf{impactDA}(\alpha) \leq r^+$, and $\mathsf{tvi}((PR, PA)) \leq t$, is $\boldsymbol{\Sigma_2^P}$-complete (for both* impactDA1 *and* impactDA2*). Moreover if $c = \infty$ the problem is* **NP***-complete (for both* impactDA1 *and* impactDA2*)*

5.5.4 MILP Formulations for Bi-Optimization Problem and Optimal Defender Strategy

The constraints defining the bi-objective optimization problem presented earlier may be non-linear. This section shows that we can solve the optimization problems in Algorithm 1 and in Section 5.5.3 via mixed integer linear programs (MILPs for short).

Variables

We first specify the principal variables in our MILP formulation.

- $x_1, \ldots, x_{|\mathbf{PR}|}$ are binary variables. Intuitively, $x_i = 0$ means product pr_i is deactivated while $x_i = 1$ means it is activated.
- $y_1, \ldots, y_{|\mathbf{PA}|}$ are binary variables. Intuitively, $y_i = 0$ means patch pa_i is not applied while $y_i = 1$ means it is applied.
- $d_1, \ldots, d_{|\mathbf{V}|}$ are real variables in $[0, 1]$. If vulnerability v_i is unprotected, then $d_i = 1$ — otherwise, $d_i = 0$.
- $dx_1, \ldots, dx_{|\mathbf{V}|}$ are real variables in $[0, 1]$. If at least one product in $PR(v_i)$ is activated, then $dx_i = 1$; otherwise, $dx_i = 0$.
- $dy_1, \ldots, dy_{|\mathbf{V}|}$ are real variables in $[0, 1]$. If no patches in $PA(v_i)$ are applied, then $dy_i = 1$; otherwise, $dy_i = 0$.
- $dd_1, \ldots, dd_{|\mathbf{V}|}$ are real variables in $[0, 1]$. If there is at least one unprotected vulnerability in $in(G, v_i)$, then $dd_i = 1$; otherwise, $dd_i = 0$.

Basic Constraints

1. **Product constraints:** If at least one product in $PR(v_i)$ is activated, then $dx_i = 1$; otherwise, $dx_i = 0$. Thus, we introduce the following constraints:

$$\begin{array}{rl} x_j \leq dx_i & v_i \in \mathbf{V}, pr_j \in PR(v_i) \\ \sum_{pr_j \in PR(v_i)} x_j \geq dx_i & v_i \in \mathbf{V} \\ 0 \leq dx_i \leq 1 & v_i \in \mathbf{V} \end{array}$$

   The first constraint captures the fact that dx_i is set to one if at least one product in $PR(v_i)$ is activated. Intuitively, $dx_i = 1$ means that vulnerability v_i is active. The second constraint says dx_i cannot be made 1 when no product in $PR(v_i)$ is activated — it forces dx_i to be zero in this case.
2. **Patch constraints:** If no patches in $PA(v_i)$ are applied, then $dy_i = 1$; otherwise, $dy_i = 0$. Thus, we introduce the following constraints:

$$\begin{array}{rl} y_j \leq 1 - dy_i & v_i \in \mathbf{V}, pa_j \in PA(v_i) \\ \sum_{pa_j \in PA(v_i)} y_j \geq 1 - dy_i & v_i \in \mathbf{V} \\ 0 \leq dy_i \leq 1 & v_i \in \mathbf{V} \end{array}$$

   These constraints are similar to those in the preceding item but apply to patches. The idea is to ensure that dy_i is set to 1 when no patches

in $PA(v_i)$ are applied. The first constraint says that if a patch pa_j for vulnerability v_i is applied, then dy_i must be set to 0 (which represents the fact that at least one patch in $PA(v_i)$ is applied). The second constraint says when no patches for vulnerability v_i are applied then dy_i must be set to 1 (the converse of the previous sentence).

3. **Unprotected vulnerabilities and incoming edges:** If there is at least one unprotected vulnerability in $in(G, v_i)$, then $dd_i = 1$; otherwise, $dd_i = 0$. Thus, we introduce the following constraints:

$$\begin{aligned} d_j &\leq dd_i & v_i \in \mathbf{V}, v_j \in in(G, v_i) \\ \textstyle\sum_{v_j \in in(v_i)} d_j &\geq dd_i & v_i \in \mathbf{V} \\ 0 \leq dd_i &\leq 1 & v_i \in \mathbf{V} \end{aligned}$$

The constraints capture the cases when at least one vulnerability is unprotected by analyzing the graph structure of the VDG we are considering. The first constraint says that when there is an edge from v_j to v_i in the VDG and v_j is unprotected (i.e. $d_j = 1$) then dd_i is also 1 as there is at least one incoming edge to v_i from a vertex (v_j) which is unprotected. The second constraint is the converse of the first, establishing an if and only if relationship.

4. **Unprotected vulnerabilities:** If vulnerability v_i is unprotected, then $d_i = 1$; otherwise, $d_i = 0$. Thus, we introduce the following constraints:

$$\begin{aligned} dx_i &\geq d_i & v_i \in \mathbf{V} \\ dy_i &\geq d_i & v_i \in \mathbf{V} \\ dd_i &\geq d_i & v_i \in \mathbf{V} \setminus S(G) \\ dx_i + dy_i + dd_i &\leq 2 + d_i & v_i \in \mathbf{V} \setminus S(G) \\ dx_i + dy_i &\leq 1 + d_i & v_i \in S(G) \end{aligned}$$

The first constraint here states that if vulnerability v_i is unprotected ($d_i = 1$) then at least one product in $PR(v_i)$ must be activated (otherwise the vulnerability would not need protection). The second constraint says that if vulnerability v_i is unprotected ($d_i = 1$) then no patch for it is activated. The third constraint says that if vulnerability v_i is unprotected ($d_i = 1$) and v_i is not a source vertex in the VDG, then one of its incoming edges is from a vertex that is unprotected. The fourth constraint says that when at least one product in $PR(v_i)$ is activated and no patch for v_i is activated and v_i is a non-source vertex that has an incoming edge from an unprotected vulnerability, then v_i is unprotected. The fifth constraint is similar but applies to vulnerabilities that are sources of the VDG.

Together, the above constraints provide basic constraints that capture the dependencies between the variables in the mixed integer linear program.

Defender Strategy in the General Case

We now specify the mixed integer linear program for the defender in the general case when no assumptions are made. We show the two integer linear

programs where $\mathsf{impactDA1}, \mathsf{impactDA2}$ measure the impact of the attacker's strategy. Note that we will use an arbitrary small number ϵ to model constraints involving the strict less than operator $<$.

$$getp(r^+, \hat{c}, c, u) = \textbf{minimize} \quad - \sum_{pr_j \in \mathbf{PR}} \mathsf{Prod}(pr_j) \cdot x_j \tag{5.6}$$

subject to

$$(1 - |\alpha|) \cdot impact(v_i) + impact(v_i) \cdot \sum_{v_j \in \alpha} d_j \leq r^+ - \epsilon \qquad \alpha \in PAS(c, u), v_i \in \alpha \tag{5.7}$$

$$\text{basic constraints hold} \tag{5.8}$$

$$\sum_{pa_j \in \mathbf{PA}} \mathsf{CostD}(pa_j) \cdot y_j \leq \hat{c} \tag{5.9}$$

$$0 \leq d_i \leq 1 v_i \in \mathbf{V} \tag{5.10}$$

$$y_i \in \{0, 1\} \qquad v_i \in \mathbf{V} \tag{5.11}$$

$$x_j \in \{0, 1\} \qquad pr_j \in \mathbf{PR} \tag{5.12}$$

As the variables dx_i, dy_i and dd_i (for all $v_i \in \mathbf{V}$) only need to be in $[0, 1]$, we do not require these to be explicitly constrained above.[4] The objective function in the above formulation says we want to minimize the negative of the productivity which is equivalent to maximizing the productivity. Constraint 5.8 represents a *set* of constraints. The defender has no idea which of the possible attacks $\alpha \in PAS(c, u)$ the attacker will choose. He needs to defend against all possible attacks. Therefore, for each attack $\alpha \in PAS(c, u)$ and each $v_i \in \alpha$, he writes a constraint of the form 5.8. saying that the impact of each vulnerability in α must be strictly less than r^+ — the strict less than requirement is captured as $r^+ - \epsilon$ for an arbitrarily large ϵ in this constraint (when $\mathsf{impactDA1}$ is used). The left side of the constraint describes the impact of the vulnerabilities in α. The second expession $\Sigma_{v_j \in \alpha} d_j$ is the number of unprotected vulnerabilities in α. If this sum equals $|\alpha|$, then α is a valid attack and in this case, the first term $(1 - |\alpha|) \cdot impact(v_i)$ makes the left hand side $(1 - |\alpha|) \cdot impact(v_i) + impact(v_i) \cdot \sum_{v_j \in \alpha} d_j$ evaluate to $impact(v_i)$ and by the semantics of $\mathsf{impactDA1}$, the constraint merely states that $\mathsf{Impact}(v_i) \leq r^+ - \epsilon$.

In order to model the min max in the MILP $getr(p^+, \hat{c}, c, u)$ we use another real variable denoted by h.

[4] Due to the nature of the constraints, ds_i, dy_i, dd_i are all guaranteed to have only $\{0, 1\}$ values even though we do not explicitly include constraints for this.

$$getr(p^+, \hat{c}, c, u) = \quad \textbf{minimize} \;\; h \tag{5.13}$$

subject to

$$(1 - |\alpha|) \cdot impact(v_i) + impact(v_i) \cdot \sum_{v_j \in \alpha} d_j \leq h \qquad \alpha \in PAS(c,u), v_i \in \alpha \tag{5.14}$$

$$\text{basic constraints hold} \tag{5.15}$$

$$\sum_{pa_j \in \mathbf{PA}} \mathsf{CostD}(pa_j) \cdot y_j \leq \hat{c} \tag{5.16}$$

$$\sum_{pr_j \in \mathbf{PR}} \mathsf{Prod}(pr_j) \cdot x_j = p^+ \tag{5.17}$$

$$0 \leq d_i \leq 1 \qquad v_i \in \mathbf{V} \tag{5.18}$$

$$y_i \in \{0,1\} \qquad v_i \in \mathbf{V} \tag{5.19}$$

$$x_j \in \{0,1\} \qquad pr_j \in \mathbf{PR} \tag{5.20}$$

$$h \geq 0 \tag{5.21}$$

The intuition behind constraint 5.14 is similar to that in the MILP definition $getp()$ earlier. The major difference is that instead of bounding the impact of the unprotected vulnerabilities on the right hand side of this constraint, we are setting it to an unknown value h and then requiring the objective function to minimize h (which captures the impact of the different attacker strategies $\alpha \in PAS(c,u)$.

Important Note

An important note in the MILPs for $getp()$ and $getr()$ is the following. In the Pareto Frontier algorithm, $getp()$ is first invoked in line (3) with $r^+ = \infty$ to return a value of p^+. Later, when r^+ is invoked in line 5, it is invoked with the value of p^+. Subsequently, when p^+ is computed again in Line 7, it is invoked with the value of r^+ computed in line 5 and this is repeated for every execution of the while loop (lines $4-8$). This explains how these two MILPs work in synchrony with the Pareto Frontier algorithm to generate the desired results.

Using impactDA2 instead of impactDA1

If we choose to use the second measure of impact of a set of vulnerabilities, we need to replace Constraint 5.7 and Constraint 5.14 with the following constraints that capture the intuition that the impact of α is the summation of the impacts of the vulnerabilities in α (as opposed to the max which is used in impactDA1):

$$(1 - |\alpha|) \cdot \sum_{v_i \in \alpha} impact(v_i) + \sum_{v_j \in \alpha} impact(v_j) \cdot \sum_{v_s \in \alpha} d_s \leq r^+ - \epsilon \quad \alpha \in PAS(c,u)$$

and

$$(1-|\alpha|)\cdot\sum_{v_i\in\alpha} impact(v_i)+\sum_{v_j\in\alpha} impact(v_j)\cdot\sum_{v_s\in\alpha} d_s\leq h\quad \alpha\in PAS(c,u).$$

The ILP version of the best defender strategy showed in Section 5.5.3 is reported in the following:

$$\textbf{minimize}\sum_{v_i\in\mathbf{V}}\mathsf{Impact}(v_i)\cdot d_i \tag{5.22}$$

subject to

$$(1-|\alpha|)\cdot impact(v_i)+impact(v_i)\cdot\sum_{v_j\in\alpha} d_j\leq r^+ \qquad \alpha\in PAS(c,u),v_i\in\alpha \tag{5.23}$$

$$\text{basic constraints hold} \tag{5.24}$$

$$\sum_{pa_j\in\mathbf{PA}}\mathsf{CostD}(pa_j)\cdot y_j\leq\hat{c} \tag{5.25}$$

$$\sum_{pr_j\in\mathbf{PR}}\mathsf{Prod}(pr_j)\cdot x_j=p^+ \tag{5.26}$$

$$0\leq d_i\leq 1 \qquad v_i\in\mathbf{V} \tag{5.27}$$

$$y_i\in\{0,1\} \qquad v_i\in\mathbf{V} \tag{5.28}$$

$$x_j\in\{0,1\} \qquad pr_j\in\mathbf{PR} \tag{5.29}$$

$$h\geq 0 \tag{5.30}$$

This formulation is a combination of constraints of the two previous ones where the objective function represents $\mathsf{tvi}((PR,PA))$. Since this formulation is a combination, in the following we only focus on the previous ones.

A critical point in these formulations is the exponential number of constraints (Constraints 5.8 and 5.14) due to the set $PAS(c,u)$. We use a row generation technique to avoid enumerating all the exponential number of constraints at the very beginning.

Row Generation (RG) Technique

RG is used for problems with a huge number of constraints where the complete enumeration of all constraints is not possible (see [137, 138]). Once such constraints are identified (in our case, these are the Constraints 5.8 and Constraints 5.14), the technique starts with a particular "restricted program". The restricted program is obtained from the MILP in Equations 5.6-5.12 (Program in Equations 5.13-5.21) by removing Constraints 5.8 (resp. Constraints 5.14). The technique iteratively performs the following two steps:

1. solve the current restricted program obtaining a solution that it is called restricted solution.
2. given the restricted solution, find an unsatisfied constraint (not present in the restricted program) w.r.t. the restricted solution, and

- if such a constraint exists, add the constraint to the restricted program and continue the iterations.
- otherwise, stop the iterations and return the current restricted solution — this is guaranteed to be the best solution for the non restricted program.

We focus only on impactDA1 (as impactDA2 is similar), and we define two methods $getpR$ and $getrR$ that represents the generic restricted programs. The method $\delta^r = getpR(r^+, \hat{c}, c, u, PAS')$ takes as input, the same parameter of $getp$ plus a set PAS'. This method solves the Program in Equations 5.6-5.12 where the Constraints 5.8 are changed with the following set of constraints depending to PAS'

$$(1 - |\alpha|) \cdot impact(v_i) + impact(v_i) \cdot \sum_{v_j \in \alpha} d_j \leq r^+ - \epsilon \qquad \alpha \in PAS', v_i \in \alpha$$

and returns the defender strategy δ^r (the restricted solution) represented by the variables of type x and y. Instead, the method

$$(\delta^r, h^r) = getpR(p^+, \hat{c}, c, u, PAS')$$

takes as input the same parameters of $getp$ plus a set PAS''. This method solves the Program in Equations 5.13-5.21) where the Constraints 5.14 are changed with the following set of constraints depending on PAS'':

$$(1 - |\alpha|) \cdot impact(v_i) + impact(v_i) \cdot \sum_{v_j \in \alpha} d_j \leq h \qquad \alpha \in PAS'', v_i \in \alpha$$

and returns the defender strategy δ^r (the restricted solution) represented by the variables of type x and y, and h^r the optimal value of h.

Note that, calling the method $getpR(p^+, \hat{c}, c, u, \emptyset)$ ($getrR(p^+, \hat{c}, c, u, \emptyset)$) is equivalent to solving the restricted program without Constraints 5.8 (Constraints 5.14).

At this point we need to find a constraint or some constraints that do not satisfy the current restricted solution provided from the restricted program. Let δ^r be the current restricted solution. Then this problem is equivalent to finding an attack strategy $\alpha \in PAS_{\delta^r}(c, u)$ (then valid for δ^r) such that $\max_{v_i \in \alpha} impact(v_i)$ is greater than a certain threshold q. For $getp$ the value of q will be $r^+ - \epsilon$, instead for the program $getr$ the value of q will be h^r (returned by $getrR()$). Then we define a method called $rowGen$ that solve the previous problem. This method takes in input δ^r and q, and returns α when it exists and $null$ otherwise. Now, the RG techniques are reported in Algorithm 2.

The result below ensures that Algorithm 2 is correct.

Proposition 5.28 *Algorithm 2 is correct and ends in a finite number of steps.*

Algorithm 2 RG techniques

```
1: procedure δ* = getpRG(r+, ĉ, c, u)
2:     PAS' = ∅
3:     δ^r = getpR(r+, ĉ, c, u, PAS');
4:     α = rowGen(δ^r, r+ − ε);
5:     while (α ≠ null) do
6:         δ^r = getpR(r+, ĉ, c, u, PAS');
7:         α = rowGen(δ^r, r+ − ε);
8:         PAS' = PAS' ∪ {α};
9:     end while
10:    return δ^r;
11: end procedure

12: procedure δ* = getrRG(p+, ĉ, c, u)
13:    PAS' = ∅
14:    (δ^r, h^r) = getrR(p+, ĉ, c, u, PAS');
15:    α = rowGen(δ^r, h^r);
16:    while (α ≠ null) do
17:        (δ^r, h^r) = getrR(p+, ĉ, c, u, PAS');
18:        α = rowGen(δ^r, h^r);
19:        PAS' = PAS' ∪ {α};
20:    end while
21:    return δ^r;
22: end procedure
```

Proof. We first prove that Algorithm 2 ends in a finite number of steps. At each iteration, the algorithm adds a constraint not present in the restricted program (if the algorithm added a constraint already present in the restricted program, then the restricted solution would not be feasible). Therefore, in the worst case, the algorithm adds all constraints and then ends. Since we have a finite number of constraints, the algorithm ends in a finite number of iterations. To prove that Algorithm 2 is correct, we observe that the optimal value reached at each iteration from the restricted program represents a lower bound of the optimal value reached from the formulation with all constraints. If Algorithm 2 stops, then there is no constraint that is not satisfied by the current restricted solution. Therefore, the current restricted solution satisfies all constraints and, then, it is also the best solution for the program.

Discussion on Correctness

Before describing the implementation of the row generation technique, we briefly step aside to review where we are.

1. The natural formulation of the defender's problem as a Pareto optimization problem involves non-linearities as shown in Sections 5.5.1 through

5.5.3. In particular, the defender's problem is a non-linear integer Pareto optimization problem.

2. In the next step (Section 5.5.4), we showed how to represent the defender's problem as a mixed integer linear programming problem removing the non-linearity in the original Pareto formulation.
3. Finally, we used row generation in Algorithm 2 to avoid enumerating all the integer linear constraints above at the very beginning, instead of taking the huge amount of time required to explicitly enumerate all needed constraints.

Proposition 5.28 shows that this sequence of steps is correct and that the best solution of the original Pareto optimization problem coincides with the best solution obtained after performing these three steps. Collectively, the second and third steps provide two big speedups of the problem faced by the defender.

Implementation of the RG technique

The effectiveness of a row generation technique depends on the total number of the generated constraints. In order to reduce this number, it is important that α, returned by $rowGen(\delta^r, q)$, is also valid for different defender strategies and not only for δ^r. A heuristic strategy minimizes the number of vulnerabilities in α (i.e. its cardinality). In fact, the smaller the cardinality of α, the greater is the possibility that α is a valid attacker strategy for another defender strategy. Moreover this is also effective because we consider the case with $\mathsf{impactDA1}(\alpha) = \max_{v \in \alpha} \mathsf{Impact}(v)$ and then its value is less dependent from the cardinality of α (differently from $\mathsf{impactDA2}(\alpha) = \sum_{v \in \alpha} \mathsf{Impact}(v)$).

Thus, we formalize this problem in the following way:

$$\alpha^* = arg\,max_{\alpha \in PAS_{\delta^r}(c,u), \left(\max_{v_i \in \alpha} impact(v_i) > q\right)} |\alpha|$$

We can compute this by an Integer Linear Program (ILP), formalized in the following way. For each element $v_i \in UV(G, \delta^r)$ we define: (i) a binary variable $k_i \in \{0, 1\}$ that is 1 if the attacker chooses to use vulnerability v in its strategy and 0 otherwise (in the same way of Section 5.4.3), and (ii) a binary variable $c_i \in \{0, 1\}$ that we will use to model the max operator in $\max_{v_i \in \alpha} impact(v_i)$. Let $maxImpact = (\max_{v_i \in UV(G,\delta^r)} impact(v_i))$. Then the ILP is the following:

$$\textbf{maximize} \quad \sum_{v_i \in UV(G,\delta^r)} k_i \tag{5.31}$$

subject to

$$impact(v_i) \cdot k_i + maxImpact \cdot c_i \geq q + eps \qquad v_i \in UV(G,\delta^r) \tag{5.32}$$

$$\sum_{v_i \in UV(G,\delta^r)} c_i = |UV(G,\delta^r)| - 1 \tag{5.33}$$

$$\sum_{v_i \in UV(G,\delta^r)} \mathsf{utilA}(v) \cdot k_i \geq u \tag{5.34}$$

$$\sum_{v_i \in UV(G,\delta^r)} \mathsf{costA}(v_i) \cdot k_i \leq c \tag{5.35}$$

$$k_i - \sum_{v_j \in in(G,v_i)} k_j \leq 0 \qquad v_i \in UV(G,\delta^r) \setminus S(G) \tag{5.36}$$

$$k_i \in \{0,1\} \qquad v_i \in UV(G,\delta^r) \tag{5.37}$$

$$c_i \in \{0,1\} \qquad v_i \in UV(G,\delta^+) \tag{5.38}$$

where the optimization function represents the cardinality of α. Constraints 5.32-5.33 model the nonlinear constraint $\max_{v_i \in \alpha} impact(v_i) > q$ and Constraints 5.34-5.35- 5.36 model the fact that $\alpha \in PAS_{\delta^r}(c,u)$.

Defender Strategy with Unbounded Cost

By Proposition 5.17 we capture the defender's optimal strategy by using MILP with a polynomial number of constraints. We use the real variable h to model the min max. We also use a binary variable z — intuitively $z = 1$ means $\sum_{v \in UV(G,\delta)} \mathsf{utilA}(v) \geq u$. Then the optimization problems for the Pareto analysis in the defender strategy where $\mathsf{impactDA} = \mathsf{impactDA1}$ are:

$$p(r^+, \hat{c}, u) = \quad \textbf{minimize} \quad - \sum_{pr_j \in \mathbf{PR}} \mathsf{Prod}(pr_j) \cdot x_j \tag{5.39}$$

subject to

$$h - (\sum_{v_i \in \mathbf{V}} \mathit{impact}(v_i)) \cdot (1 - z) \leq r^+ - \epsilon \tag{5.40}$$

$$\mathit{impact}(v_i) \cdot d_i \leq h \qquad v_i \in \mathbf{V} \tag{5.41}$$

$$\text{basic constraints hold} \tag{5.42}$$

$$\sum_{pa_j \in \mathbf{PA}} \mathsf{CostD}(pa_j) \cdot y_j \leq \hat{c} \tag{5.43}$$

$$\sum_{v_i \in \mathbf{V}} \mathsf{utilA}(v_i) \cdot d_i - (\sum_{v_i \in \mathbf{V}} \mathsf{utilA}(v_i)) \cdot z \leq u - \epsilon \tag{5.44}$$

$$0 \leq d_i \leq 1 \qquad v_i \in \mathbf{V} \tag{5.45}$$

$$y_i \in \{0, 1\} \qquad v_i \in \mathbf{V} \tag{5.46}$$

$$x_j \in \{0, 1\} \qquad pr_j \in \mathbf{PR} \tag{5.47}$$

$$z \in \{0, 1\} \tag{5.48}$$

$$h \geq 0 \tag{5.49}$$

The second MILP uses another real variable $h1$ such that when $z = 1$ ($\sum_{v \in UV(G,\delta)} \mathsf{utilA}(v) \geq u$) we have $h = h1$ — otherwise it assumes in the minimization value 0.

$$r(p^{+}, \hat{c}, u) = \quad \textbf{minimize} \;\; h1 \tag{5.50}$$

subject to

$$h - h1 - (\sum_{v_i \in \mathbf{V}} impact(v_i)) \cdot (1 - z) \leq 0 \tag{5.51}$$

$$impact(v_i) \cdot d_i \leq h \qquad v_i \in \mathbf{V} \tag{5.52}$$

$$\text{basic constraints hold} \tag{5.53}$$

$$\sum_{pa_j \in \mathbf{PA}} \mathsf{CostD}(pa_j) \cdot y_j \leq \hat{c} \tag{5.54}$$

$$\sum_{pr_j \in \mathbf{PR}} \mathsf{Prod}(pr_j) \cdot x_j = p^{+} \tag{5.55}$$

$$\sum_{v_i \in \mathbf{V}} \mathsf{utilA}(v_i) \cdot d_i - (\sum_{v_i \in \mathbf{V}} \mathsf{utilA}(v_i) \cdot u \leq \mu + \epsilon \tag{5.56}$$

$$0 \leq d_i \leq 1 \qquad v_i \in \mathbf{V} \tag{5.57}$$

$$y_i \in \{0, 1\} \qquad v_i \in \mathbf{V} \tag{5.58}$$

$$x_j \in \{0, 1\} \qquad pr_j \in \mathbf{PR} \tag{5.59}$$

$$z \in \{0, 1\} \tag{5.60}$$

$$h \geq 0 \tag{5.61}$$

$$h1 \geq 0 \tag{5.62}$$

When we consider the case where we measure impact of an attacker strategy via the function impactDA2, we need to change Constraint 5.41 and Constraint 5.52 to:

$$\sum_{v_i \in \mathbf{V}} impact(v_i) \cdot d_i \leq h.$$

5.5.5 Possible Extensions

Our framework can be extended in order to accommodate slightly different scenarios. In this section we provide a few examples.

There can be relationships among different software products, expressed by constraints imposing that if a software product is deactivated, other ones will automatically be deactivated as well (because the former provides services to the latter). To represent the fact that if product pr_1 is deactivated then product pr_2 must be deactivated, it suffices to add to the basic constraints (Section 5.5.4) an additional one of the form $x_2 \leq x_1$ (where $x_i = 1$ if pr_i is activated and zero otherwise, see Section 5.5.4). Obviously, if we deactivate pr_1, the total productivity decreases of the productivity values of p_1 and p_2.

We can also represent more complex relationships defined by a generic boolean formula, i.e. impose the deactivation of a product if a boolean formula on the activation variables of other products evaluates to *true*. If we assume

that product pr_1 must be deactivated if pr_2 or pr_3 are deactivated together with pr_4 (i.e. the boolean formula for the deactivation of pr_1 is $(x_2 = 0 \vee x_3 = 0) \wedge x_4 = 0$), we add two variables $out_{2\vee 3}$ and $out_{(2\vee 3)\wedge 4}$ along with the following constraints:

$$\begin{aligned}
1 - x_2 &\leq out_{2\vee 3} \\
1 - x_3 &\leq out_{2\vee 3} \\
(1 - x_2) + (1 - x_3) &\geq out_{2\vee 3} \\
1 - x_4 &\geq out_{(2\vee 3)\wedge 4} \\
out_{2\vee 3} &\geq out_{(2\vee 3)\wedge 4} \\
(1 - x_4) + out_{2\vee 3} &\leq 1 + out_{(2\vee 3)\wedge 4} \\
1 - out_{(2\vee 3)\wedge 4} &\geq x_1 \\
0 \leq out_{(2\vee 3)\wedge 4} &\leq 1 \\
0 \leq out_{2\vee 3} &\leq 1
\end{aligned}$$

The same approach can be used to model relationships among patches, i.e. to impose that a patch can only be applied if other patches are applied.

Other possible extensions involve the definition of the total productivity and cost. For instance, we could compute the productivity of a set of products by taking the maximum (or the minimum) of their productivities. Assume that we have 3 products pr_1, pr_2, and pr_3 and want to take the maximum of their productivities. We then add three boolean variables m_1, m_2, and m_3 and a real variable representing the total productivity $prod$, along with the following constraints:

$$\begin{aligned}
x_1 \cdot \mathsf{Prod}(pr_1) &\leq prod \\
x_2 \cdot \mathsf{Prod}(pr_2) &\leq prod \\
x_3 \cdot \mathsf{Prod}(pr_3) &\leq prod \\
x_1 \cdot \mathsf{Prod}(pr_1) + (1 - m_1) * \max\{\mathsf{Prod}(pr_1), \mathsf{Prod}(pr_2), \mathsf{Prod}(pr_3)\} &\geq prod \\
x_2 \cdot \mathsf{Prod}(pr_2) + (1 - m_2) * \max\{\mathsf{Prod}(pr_1), \mathsf{Prod}(pr_2), \mathsf{Prod}(pr_3)\} &\geq prod \\
x_3 \cdot \mathsf{Prod}(pr_3) + (1 - m_3) * \max\{\mathsf{Prod}(pr_1), \mathsf{Prod}(pr_2), \mathsf{Prod}(pr_3)\} &\geq prod \\
m_1 + m_2 + m_3 &= 2 \\
m_1, m_2, m_3 &\in \{0, 1\} \\
prod &\geq 0
\end{aligned}$$

The same approach can be used to model the case where we take the maximum (or the minimum) patch cost.

In another possible scenario, we have the same product installed on k different machines, so we may need to patch the same vulnerability on each of such machines. The patching cost is not necessarily equal to k times the cost of the patch — instead it could be provided by a generic function $fcost$. In this case, we can add k new boolean variables $l_1, \ldots, l_k$ and the following constraints:

$$\begin{array}{ll}
\sum_{i=1}^{k} y_i \geq l_{sum} * sum & sum \in \{0,\ldots,n\} \\
\sum_{i=1}^{k} y_i \leq k(1-l_{sum}) + l_{sum} * sum & k \in \{0,\ldots,n\} \\
\sum_{i=1}^{k} l_i = 1 & \\
l_{sum} \in \{0,1\} & sum \in \{0,\ldots,n\}
\end{array}$$

then define the total cost as $\sum_{i=1}^{k} l_i \cdot fcost(i)$. The same approach can be used to model the case where we use a generic function that computes productivity values based on the number of activated products.

5.6 Experimental Results

In this section we report on the experimental results of the proposed framework when applied to a number of realistic scenarios. We implemented the framework in Java and used IBM ILOG CPLEX 12.5 for solving the integer linear programs. All the experiments were run on an Intel Core i7-3770K CPU clocked at 3.50GHz, running Windows 8, with 10GB RAM available for the experiments.

We applied the proposed techniques to 4 different VDGs, whose main statistics are reported in Fig. 5.6. VDG1 was built from the attack graph in [139] according to the translation technique described in the next subsection, and VDG2 is an extension of VDG1 where we added 3 subnetworks having a similar structure as the original ones. VDG4 was extracted from the dataset in [140] that represents a network whose topology follows the *power-law* rule — we assumed that each edge connects two vulnerabilities in our scenario. Finally, VDG3 is a subgraph extracted from VDG4. The vulnerabilities we considered in VDG1 and VDG2 were exactly those present in the original attack graphs — we derived their associated products, patches, and impact values directly from the NVD. Finally, we associated each vertex in VDG1 and VDG2 with a vulnerability randomly extracted from the NVD. Additional details about the topology of the VDGs are reported in the next subsection.

	$\lvert\mathbf{V}\rvert$	$\lvert E\rvert$	$\lvert S(G)\rvert$	$\lvert\mathbf{PA}\rvert$	$\lvert\mathbf{PR}\rvert$
VDG1	249	3,496	65	1,191	4,665
VDG2	900	20,556	144	4,011	13,720
VDG3	2,500	5,390	476	15,286	81,686
VDG4	10,000	30,627	1,400	62,065	284,089

Fig. 5.6. Main statistics of the VDGs used in the experiments

5.6.1 Topology of the VDGs Used in the Experiments

In the experimental evaluation we used 4 different vulnerability dependency graphs, whose main statistics are reported in Fig. 5.6. In particular, VDG1

was built by translating the real-world attack graph in [139] according to the translation technique described above. The overall structure of this attack graph is reported in Fig. 5.7. The graph contains 3 trust partitions, each corresponding to a different subnetwork (128, 130, and 140). The figure reports in parentheses, for each trust partition, the number of vulnerabilities that allow to compromise a machine in the partition (e.g., 14 vulnerabilities in subnetwork 128). The figure also reports the number of vulnerabilities that are associated to movement edges between subnetworks (e.g., 45 vulnerabilities on the movement edge from subnetwork 128 to subnetwork 140). The resulting VDG has 249 vertices and 3,496 edges.

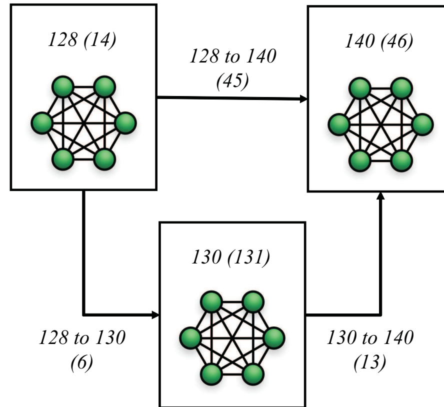


Fig. 5.7. Attack graph from [139] used to build VDG1

VDG2 was built by translating an attack graph obtained by extending the one in Fig. 5.7 with 3 additional trust partitions (corresponding to subnetworks 150, 160, and 170). Such subnetworks have a similar structure as the original ones in Fig. 5.7. The resulting attack graph, and the number of vulnerabilities considered for each trust partition and each movement edge, are shown in Fig. 5.8. The resulting VDG has 900 vertices and 20,556 edges.

VDG3 and VDG4 represent completely different scenarios. First, we wanted to evaluate how the framework behaves when facing much larger networks. Second, we wanted to consider networks whose topology follows the *power-law* rule, i.e. most nodes have few edges, whereas a tiny fraction of nodes have a large number of edges (more formally, the fraction of nodes with L edges is proportional to L^{-k} where k is a network dependent constant). This structure is one of the main reasons why most Internet networks exhibit a very high stability and resiliency, yet they are prone to occasional collapse [141]. We therefore built VDG4 from the dataset in [140] that is a snapshot of the Gnutella peer-to-peer file sharing network, and assumed that each edge connects two vulnerabilities. The resulting VDG has 10,000 vertices and 30,627

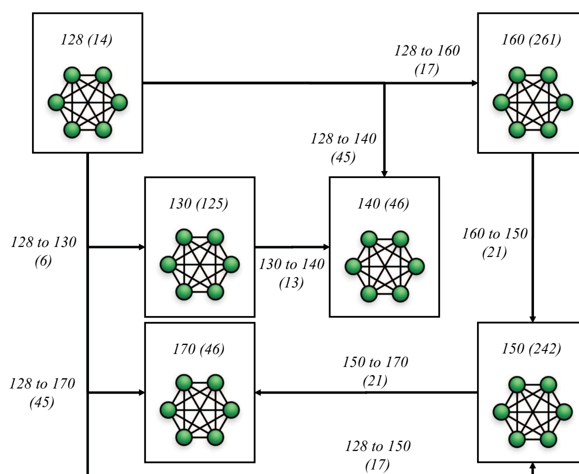


Fig. 5.8. Extension of the attack graph from [139] used to build VDG2

edges. Moreover, we selected a subset of the network with fewer vertices and edges (2,500 and 5,390, respectively) to build VDG3.

5.6.2 Pareto Frontiers

We started by computing the Pareto frontiers for VDG1 and VDG2, using both the impactDA functions, by varying the maximum patching cost $\hat{c}$ for the defender. We also applied the RG technique of Section 5.5.4 in the cases where we assumed minimum utility and maximum cost for the attacker (we fixed them to 1 and $0.1\%|\mathbf{V}|$, respectively). In these experiments, we set $\mathsf{Prod}(pr) = 1$ for all products pr, and $\mathsf{CostD}(pa) = 1$ for all patches pa. The Pareto frontiers we obtained are reported in Fig. 5.9.

The results obtained confirm my our expectations and provide some interesting insights. First, if we look at the frontier obtained for one value of allowed patching cost $\hat{c}$, we see that the Pareto points get closer to the maximum possible total productivity value — that is equal to $|\mathbf{PR}|$ because the productivity value of each product is 1 — when we increase the maximum impact allowed. When assuming no patching at all ($\hat{c} = 0$) the framework identifies the products that need to be deactivated to protect the system, while maximizing the productivity of the remaining products. For instance, using impactDA1 on VDG1 without assumptions on the attacker (Fig. 5.9(c)) the framework finds out that the system can reach a total productivity equal to 3,381, i.e. 72.5% of the maximum possible value even in the "extreme" case where the user wants to avoid any impact (maximum impact equal to zero). In addition, the maximum productivity value is reached more quickly when we increase the allowed patching cost. For instance, in Fig. 5.9(c), the standard

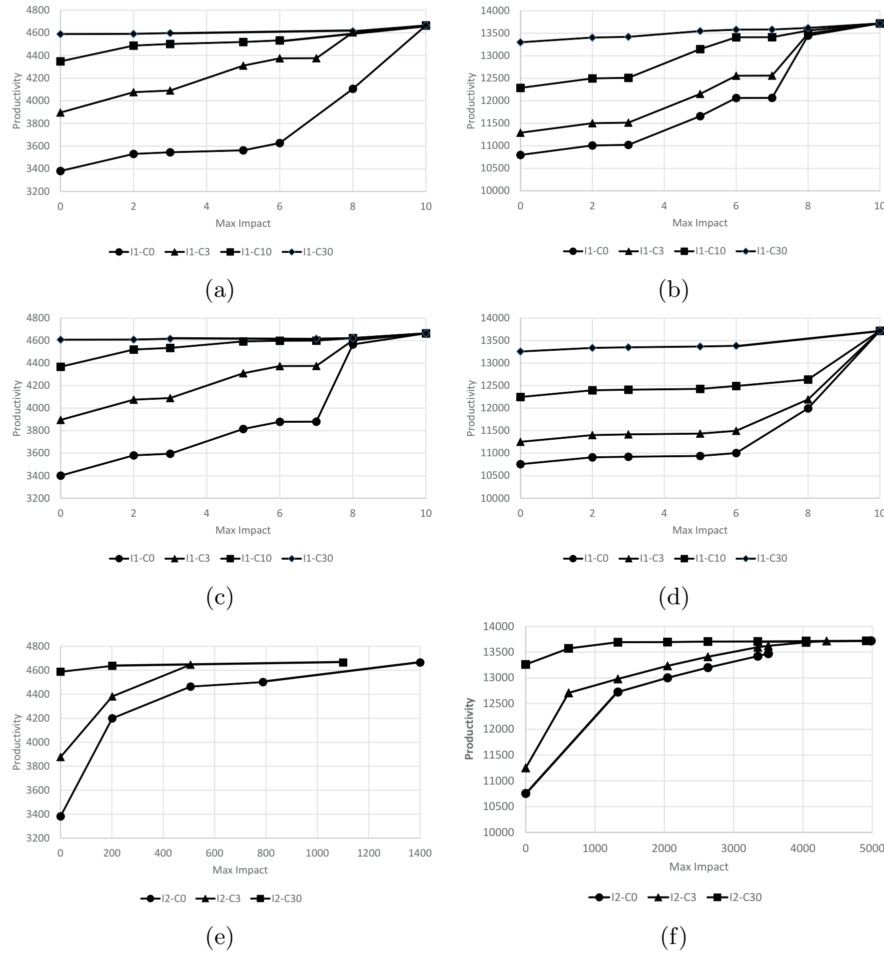


Fig. 5.9. Pareto frontiers obtained for VDG1 (left) and VDG2 (right). Label "Ix-Cy" means we used impactDAx with $\hat{c} = y$. Charts (a) and (b) represent the case where RG was applied.

deviations of the productivity values obtained for $\hat{c} = 1$, $\hat{c} = 10$, and $\hat{c} = 30$, are 420.23, 92.63, and 29.33, respectively.

In some cases, we also notice that increasing the maximum impact does not actually improve total productivity. For instance, in Fig. 5.9(c), the Pareto points obtained with $\hat{c} \in \{0, 3\}$ when the maximum impact is 2 are relatively close to those obtained with a maximum impact of 6. This provides important information to the user in order to choose a point that better corresponds to the desired tradeoff between costs and benefits: moving the maximum impact

from 2 to 6 would not provide actual benefits in terms of productivity, whereas just moving from 6 to 8 would allow a notable improvement.

Moreover, the immediate consequence of assuming a minimum utility and a maximum cost for the attacker (Figs. 5.9(a) and 5.9(b)) is that the productivity values obtained with the same allowed cost and impact for the defender are higher — obviously, at the price of the additional risks brought by making specific assumptions about the attacker. Also, as we will show in the following, the application of the RG technique to this case allows us to compute the Pareto frontiers with acceptable computational effort.

Finally, the results obtained using impactDA2 (Figs. 5.9(e) and 5.9(f)) have much higher values of maximum impact. This is an obvious consequence of the fact that we are now taking the sum of the impacts of the unprotected vulnerabilities. In addition, a much larger number of Pareto points are generated (e.g., 2,264 points for VDG2 with $\hat{c} = 0$). Only a subset of the points are represented in the figures — in general, with impactDA2, it is necessary to extract of a subset of the generated points in order to interpret the relationships among them. Moreover, taking the sum of impact values somehow makes the importance of the vulnerabilities left unprotected more difficult to assess.

In a second round of experiments, we looked at how our framework behaves when using much larger graphs. Fig. 5.10 reports the Pareto frontiers obtained for VDG3 and VDG4 with impactDA1, for different values of $\hat{c}$. The results generally confirm the observations made for VDG1 and VDG2 — for instance, for both graphs, the productivity values obtained for all values of $\hat{c}$ with a maximum impact between 0 and 4 appear very close.

Optimal Defender Strategies

In both rounds of experiments, we also computed the optimal defender strategies associated with the Pareto frontiers corresponding to $\hat{c} = 10$. An interesting result we obtained is that, while the optimization problem aims at minimizing the total impact given a fixed maximum impact, in all cases we obtained optimal total impact values lower than or equal to the maximum impact chosen.

Varying Patch Costs and Productivity Values

To assess the results obtainable by the framework when patch costs and productivity values are not fixed to the same value, we ran additional experiments using the same setting of Fig. 5.9(d), except that we varied patch costs, productivity values, and both. In the experiments with varying patch costs, we randomly assigned a cost in $[0.5, 3]$ to each patch (while keeping the same total cost for the whole set of patches). We did the same to vary productivity values.

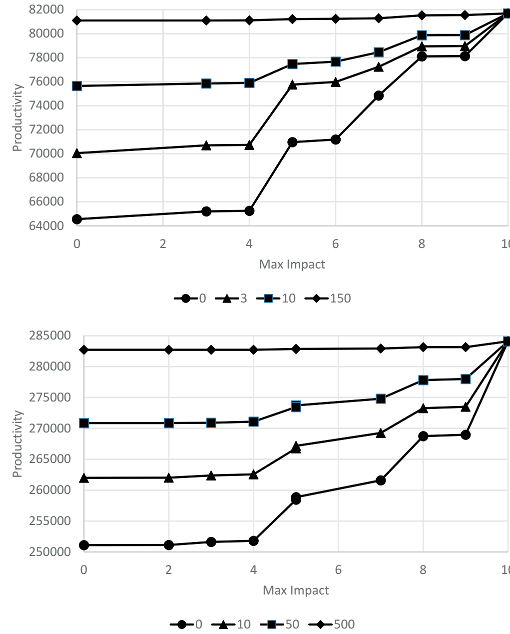


Fig. 5.10. Pareto frontiers obtained for VDG3 (left) and VDG4 (right) using impactDA1, for different values of $\hat{c}$.

Fig. 5.11 reports the results obtained when varying patch costs only. The Pareto frontier is obviously the same for maximum cost $\hat{c} = 0$, but interestingly, the total productivity values obtained for all the other values of $\hat{c}$ are consistently higher (485.9 on average). This can be explained by observing that the availability of patches with different costs corresponds to a higher flexibility in choosing the best subset of patches to apply.

Interestingly, the increase in total productivity we obtained when we also varied productivity values was much more limited (less than 1% on average). This can be explained by observing that the percentage of products that are left activated in the solutions found by the framework is always relatively high (around 80% in the worst cases, i.e. when $\hat{c} = 0$), so the increased flexibility has much less impact on total productivity.

5.6.3 Execution Time

The algorithms proposed in this chapter have an exponential worst-case time complexity, as a consequence of the reduction to integer linear programming problems. Moreover, the ILP problems we derive may include a huge number of constraints. However, tools like CPLEX embed very efficient and sophisticated heuristics that enable the solution of very large linear programs in many applications (especially when combined with RG techniques).

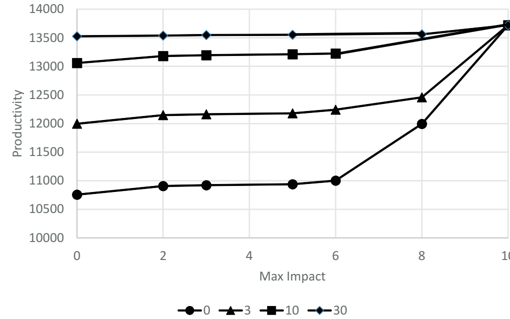


Fig. 5.11. Pareto frontiers obtained for VDG2 with impactDA1 and patch costs in $[0.5, 3]$, for different values of $\hat{c}$.

In order to assess the actual computational effort required by the proposed algorithms in realistic scenarios, we measured the time taken to compute (i) all Pareto frontiers and (ii) all optimal defender strategies associated with the Pareto frontiers corresponding to $\hat{c} = 10$.

Fig. 5.12 reports the time taken to compute a Pareto frontier for different values of $\hat{c}$ when using VDG1 and VDG2. The results show that, as expected, higher patching costs allow for more freedom in moving through the search space, and thus result in lower execution times in the majority of cases. Overall, execution times appear satisfactory: a Pareto frontier with impactDA1 was obtained in a matter of seconds for VDG1 and tens of seconds for VDG2 — the differences between the two graphs are always around 1 order of magnitude. Moreover, when moving from impactDA1 to impactDA2, execution times increase between 1 and 2 orders of magnitude. Finally, the use of the RG technique in the cases where we made assumptions about the attacker results in no substantial difference in execution time.

Finally, Fig. 5.13 reports, for all graphs, the average time taken to compute a complete Pareto frontier and the optimal defender strategy associated with a Pareto point when using impactDA1. Again, execution times appear satisfactory and acceptable for practical purposes — in addition, we can see a slightly superlinear increase with respect to the size of the graphs.

5.7 Conclusion and Future Work

Enterprise security managers are a busy lot. They are constantly called upon to fight various fires within their organization. Many organizations have underinvested in computer security personnel, leading to chronic stress for enterprise security managers. As a consequence, most enterprise security managers only patch the biggest vulnerabilities within their network [102, 103]. As a consequence, a smart attacker can easily compromise large organizations by

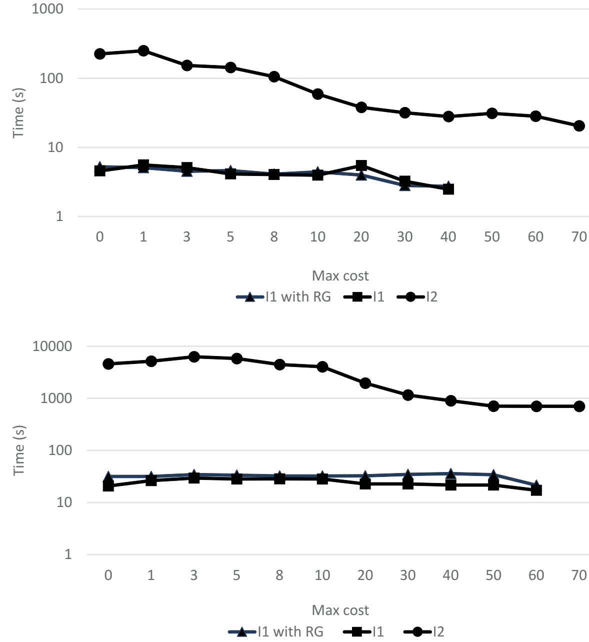


Fig. 5.12. Time taken to compute a Pareto frontier for different values of $\hat{c}$ when using VDG1 (top) and VDG2 (bottom). Label "Ix" means we used impactDAx. The y-axis is on a logarithmic scale.

guessing what commonly available software is likely used within the enterprise, and then crafting attacks that leverage vulnerabilities that have a moderate impact and are thus less likely to be patched.

In this chapter, we develop a game-theoretic model to answer the following question: *Given a finite cost that an enterprise is willing to bear to patch vulnerabilities, and given a minimal productivity level that the organization requires, what actions should the enterprise security officer take in order to best protect his enterprise?* In order to answer this question, we first model the attacker's behavior. The attacker wants to find a set of vulnerabilities to exploit whose impact (measured for instance via NIST's Common Vulnerability Scoring System) exceeds a threshold and whose cost (for him to attack) is below a threshold. We show that the problem of finding an optimal attacker strategy is NP-complete. The defender may take two kinds of actions – patching vulnerabilities (which involve a patching cost) and deactivating products (which involve removing certain products from the network which impacts productivity of the enterprise as users familiar with those products might object, might need re-training, and so forth). The defender wishes to find a set of vulnerabilities to patch so that the patching cost is below a given threshold and at the same time, he is minimizing the maximal impact the attacker can

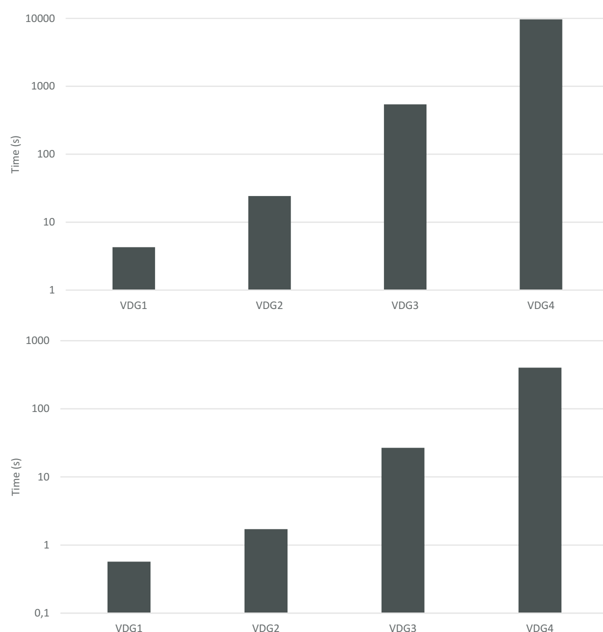


Fig. 5.13. Average time taken to compute a Pareto frontier (top) and the optimal defender strategy associated with a Pareto point with $\hat{c} = 10$ (bottom) using impactDA1. The y-axis is on a logarithmic scale.

have while simultaneously also maximizing the productivity of his enterprise. This leads to a problem involving Pareto optimization as these two goals are mutually competing. We show that checking existence of a strategy for the defender is Σ_2^P-complete and that various variants of this problem are also intractable at different levels in the polynomial hierarchy.

Then we show how to formulate the defender's problem as a convex, nonlinear optimization problem after which we show an encoding as a mixed integer *linear* optimization (MILP) problem. Because the MILP formulation involves a huge number of constraints, we show how to leverage row generation methods to solve the MILP more efficiently. We test the algorithms using two real world networks (one of which is varied in some alternative ways). We show that the Pareto optimization algorithms deliver solutions that capture a nice balance between impact of the attacker's attacks and productivity of the enterprise and that the algorithm runs in a reasonable amount of time.

5.8 Table of Symbols

The table below summarizes the symbols used in this chapter.

Symbol	*Section*	*Description*
$\mathbf{PR}$	5.3	Set of all products
$\mathbf{V}$	5.3	Set of all vulnerabilities
$\mathsf{Vuln}(pr)$	5.3	Set of all vulnerabilities of product pr
$PR(v)$	5.3	Set of products with vulnerability v
$\mathbf{PA}$	5.3	Set of all patches
$PA(v)$	5.3	Set of all patches for vulnerability v
$G = (\mathbf{V}, E)$	5.3 (Def. 5.1)	Vulnerability dependency graph
$S(G)$	5.3	Set of all vulnerabilities in G with no incoming edges
$in(G, v)$	5.3	Set of all incoming vulnerabilities of v in G
$out(G, v)$	5.3	Set of all outgoing vulnerabilities of v in G
$\mathsf{Impact}(v)$	5.3	Impact of vulnerability v for the defender
$\mathsf{CostD}(pa)$	5.3	Cost of patch pa for the defender
$Tc(PA)$	5.3	Total cost of the set PA of patches for the defender
$\mathsf{Prod}(pr)$	5.3	Productivity of product pr for the defender
$Tp(PR)$	5.3	Total productivity of the set PR of products for the defender
$\delta = (PR, PA)$	5.4.1 (Def. 5.8)	Defender strategy (set of products PR and set of patches PA)
DS	5.4.1 (Def. 5.8)	Set of all defender strategies
$UV(G, \delta)$	5.4.1	Set of all unprotected vulnerabilities of G given the defender strategy δ
$\mathsf{tvi}(\delta)$	5.4.1 (Def. 5.11)	Total vulnerability impact of the defender strategy δ
α	5.4.2 (Def. 5.13)	Attacker strategy
AS_δ	5.4.2	Set of all attacker strategies valid for the defender strategy δ
$\mathsf{impactDA1}(\alpha)$	5.4.2 (Def. 5.14)	Impact of type 1 of the attacker strategy α
$\mathsf{impactDA2}(\alpha)$	5.4.2 (Def. 5.14)	Impact of type 2 of the attacker strategy α
$\mathsf{impactDA}(\alpha)$	5.4.2	Impact of type 1 or 2 of the attacker strategy α
$\mathsf{utilA}(v)$	5.4.2	Utility of vulnerability v for the attacker
$\mathsf{costA}(v)$	5.4.2	Cost of vulnerability v for the attacker
c	5.4.2	Maximum cost for the vulnerabilities in the attacker strategy for the attacker
u	5.4.2	Minimum utility for the vulnerabilities in the attacker strategy for the attacker
$PAS(c, u)$	5.4.2 (Def. 5.16)	Preferred attacker strategies
$PAS_\delta(c, u)$	5.4.2 (Def. 5.16)	Preferred attacher strategies valid for the defender strategy δ
$\hat{c}$	5.5.1	Maximum cost for the patches in the defender strategy for the defender
(r^+, p^+)	5.5.3	Pareto point

6

Conclusions

Any organization with network-connected information systems must put appropriate countermeasures in place, in order to prevent malicious users from taking advantage of their systems. There are three important security issues to be taken care of: (*i*) making systems able to recognize and deal with ongoing attacks when a malicious user has been able to begin one; (*ii*) making systems able to identify anomalous behaviors not necessarily classifiable as attacks; (*iii*) making systems immune to known attacks. These three issues are usually referred to as *intrusion detection*, *anomaly detection*, and *adversarial defense*. This thesis has elaborated defense techniques for enterprise information systems that contribute to address these issues. In particular, a number of real-time intrusion detection techniques have been presented that, given a set of known attack patterns, index the "activities" that are happening in a monitored system in order to extract "attack instances", e.g., sub-sequences of the log that match some of the given patterns [1, 2, 3]. Moreover, an anomaly detection technique has been proposed that labels sub-sequences of the log as "unexplained" when they significantly differ from the "explained" ones [4]. Finally, a novel defense technique has been devised that, given a set of software vulnerabilities, computes the Pareto-optimal sets of vulnerabilities that have to be patched in order to cover a portion of the network as wide as possible with limited resources [5]. This technique has been developed with a security game approach, and it has been shown that there can exist multiple solutions expressed as cost-productivity pairs, i.e., there may exist more than one set of vulnerabilities to patch and software to deactivate that satisfy the security constraints, and each providing different value of patching cost and level of productivity. The feasibility and performance of the proposed techniques have been extensively validated through experimental assessments that proved their validity.

As cyber attacks continue to increase in volume and level of sophistication, future research lines should be addressed to evolve cyber-defense techniques as well. In particular future techniques should be less dependent on known

attacks, and more prone to automatic learning. It is also necessary to continuously update metrics and parameters involved in the whole security process.

The development of metrics associated with the level of security of a computer system is clearly desirable, but a clear understanding and in-depth evaluation of their limitations is mandatory as well. It is certainly possible to chronicle various existing attack strategies and ensure that a system is not vulnerable to these, but this is at best a backward-looking approach. Change detection of files and key programs can help in identifying anomalies, but correlating such anomalies to actual attacks will require further research using ideas and approaches inherited from fields such as machine learning and event processing.

In the case of techniques developed with a game-theoretic approach, a great deal of efforts are required to find solutions that best fit with real-case scenarios. Future research lines should focus on how to model a security game as an imperfect and incomplete information game, because making assumptions on attackers may lead to undesired results. Indeed, in many past works too many assumptions on attacker behaviors were groundlessly made, whereas in real scenarios an organization does not actually know whom it has to face against over time, as attackers become always more and more sophisticated. Therefore, it may result non-effective to model a security game as a complete and/or perfect information game, where is assumed the defender knows the type of the attacker or his/her utilities. On the contrary, should be taken into account the worst-case attacker (e.g., with unlimited resources) and defenders with limited costs, so that a defense strategy will result mostly oriented to the needs of the defender, instead of to the characteristics of the attacker. Henceforth, designing defense strategies against a worst-case attacker may produce best responses.

References

1. A. Guzzo, A. Pugliese, A. Rullo, D. Saccà, Intrusion detection with hypergraph-based attack models, in: Graph Structures for Knowledge Representation and Reasoning, Springer, 2014, pp. 58–73.
2. A. Guzzo, A. Pugliese, A. Rullo, D. Saccà, Hypergraph-based attack models for network intrusion detection, in: 22nd Italian Symposium on Advanced Database Systems, SEBD 2014, Sorrento Coast, Italy, June 16-18, 2014., 2014, pp. 61–68.
3. A. Pugliese, A. Rullo, Expressive and efficient online alert correlation. Submitted to IEEE Symposium on Security and Privacy 2015 conference.
4. C. Molinaro, V. Moscato, A. Picariello, A. Pugliese, A. Rullo, V. S. Subrahmanian, PADUA: parallel architecture to detect unexplained activities, ACM Trans. Internet Techn. 14 (1) (2014) 3.
5. E. serra, S. Jajodia, A. pugliese, A. Rullo, V. Subrahmanian, Pareto optimal adversarial defense of enterprise systems. Accepted by ACM Transaction on Information and System Security.
6. M. Albanese, S. Jajodia, A. Pugliese, V. S. Subrahmanian, Scalable detection of cyber attacks, in: Computer Information Systems - Analysis and Technologies - 10th International Conference, CISIM 2011, Kolkata, India, December 14-16, 2011. Proceedings, 2011, pp. 9–18.
7. A. Patcha, J.-M. Park, An overview of anomaly detection techniques: Existing solutions and latest technological trends, Computer Networks 51 (12) (2007) 3448–3470.
8. P. Garcia-Teodoro, J. E. Díaz-Verdejo, G. Maciá-Fernández, E. Vázquez, Anomaly-based network intrusion detection: Techniques, systems and challenges, Computers & Security 28 (1-2) (2009) 18–28.
9. C. Piciarelli, C. Micheloni, G. L. Foresti, Trajectory-based anomalous event detection, IEEE Trans. Circuits Syst. Video Techn. 18 (11) (2008) 1544–1554.
10. T. Shon, J. Moon, A hybrid machine learning approach to network anomaly detection, Inf. Sci. 177 (18) (2007) 3799–3821.
11. S. Kumar, E. H. Spafford, A pattern matching model for misuse intrusion detection.
12. V. Paxson, Bro: a system for detecting network intruders in real-time, Computer Networks 31 (23-24) (1999) 2435–2463.

13. G. Vigna, R. A. Kemmerer, Netstat: A network-based intrusion detection system, Journal of Computer Security 7 (1) (1999) 37–71.
14. M. Sheikhan, Z. Jadidi, Misuse detection using hybrid of association rule mining and connectionist modeling, World Applied Sciences Journal 7 (2009) 31–37.
15. M. Albanese, S. Jajodia, A. Pugliese, V. S. Subrahmanian, Scalable analysis of attack scenarios, in: Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings, 2011, pp. 416–433.
16. M. Albanese, A. Pugliese, V. S. Subrahmanian, Fast activity detection: Indexing for temporal stochastic automaton-based activity models, IEEE Trans. Knowl. Data Eng. 25 (2) (2013) 360–373.
17. A. Pugliese, V. S. Subrahmanian, C. Thomas, C. Molinaro, PASS: A parallel activity-search system, IEEE Trans. Knowl. Data Eng. 26 (8) (2014) 1989–2001.
18. R. Sadoddin, A. Ghorbani, Alert correlation survey: framework and techniques, in: Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services, ACM, 2006, p. 37.
19. C. Kruegel, F. Valeur, G. Vigna, Intrusion Detection and Correlation - Challenges and Solutions, Vol. 14 of Advances in Information Security, Springer, 2005.
20. F. Valeur, G. Vigna, C. Krügel, R. A. Kemmerer, A comprehensive approach to intrusion detection alert correlation, IEEE Trans. Dependable Sec. Comput. 1 (3) (2004) 146–169.
21. X. Ou, S. Govindavajhala, A. W. Appel, Mulval: A logic-based network security analyzer, in: 14th USENIX Security Symposium, 2005, pp. 1–16.
22. L. Wang, A. Liu, S. Jajodia, Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts, Computer Communications 29 (15) (2006) 2917–2933.
23. S. Roschke, F. Cheng, C. Meinel, A new alert correlation algorithm based on attack graph, in: Á. Herrero, E. Corchado (Eds.), CISIS, Vol. 6694 of Lecture Notes in Computer Science, Springer, 2011, pp. 58–67.
24. C.-H. Mao, H.-K. Pao, C. Faloutsos, H.-M. Lee, Sbad: Sequence based attack detection via sequence comparison, in: C. Dimitrakakis, A. Gkoulalas-Divanis, A. Mitrokotsa, V. S. Verykios, Y. Saygin (Eds.), PSDML, Vol. 6549 of Lecture Notes in Computer Science, Springer, 2010, pp. 78–91.
25. O. Sheyner, J. Haines, S. Jha, R. Lippmann, J. M. Wing, Automated generation and analysis of attack graphs, in: Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on, IEEE, 2002, pp. 273–284.
26. J.-s. Liu, R.-h. Li, Y.-l. Liu, Z.-y. ZHANG, Multi-sensor data fusion based on correlation function and fuzzy integration function, Systems Engineering and Electronics 28 (7) (2006) 1006–1009.
27. C. Michael, A. Ghosh, Using finite automata to mine execution data for intrusion detection: A preliminary report, in: Recent Advances in Intrusion Detection, Springer, 2000, pp. 66–79.
28. R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, A fast automaton-based method for detecting anomalous program behaviors, in: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on, IEEE, 2001, pp. 144–155.

29. A. P. Kosoresow, S. A. Hofmeyr, Intrusion detection via system call traces, IEEE software 14 (5) (1997) 35–42.
30. D. Wagner, D. Dean, Intrusion detection via static analysis, in: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on, IEEE, 2001, pp. 156–168.
31. A. Babenko, L. Mariani, F. Pastore, Ava: automated interpretation of dynamically detected anomalies, in: ISSTA, Vol. 9, 2009, pp. 237–248.
32. J. Branch, A. Bivens, C. Y. Chan, T. K. Lee, B. K. Szymanski, Denial of service intrusion detection using time dependent deterministic finite automata, in: Proc. Graduate Research Conference, Citeseer, 2002, pp. 45–51.
33. S. Noel, S. Jajodia, Managing attack graph complexity through visual hierarchical aggregation, in: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security, ACM, 2004, pp. 109–118.
34. P. Ning, Y. Cui, D. S. Reeves, D. Xu, Techniques and tools for analyzing intrusion alerts, ACM Trans. Inf. Syst. Secur. 7 (2) (2004) 274–318.
35. H. Ren, N. Stakhanova, A. A. Ghorbani, An online adaptive approach to alert correlation, in: C. Kreibich, M. Jahnke (Eds.), DIMVA, Vol. 6201 of Lecture Notes in Computer Science, Springer, 2010, pp. 153–172.
36. G. Creech, J. Hu, A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns.
37. T. Bass, Intrusion detection systems and multisensor data fusion, Communications of the ACM 43 (4) (2000) 99–105.
38. F. Cuppens, A. Miege, Alert correlation in a cooperative intrusion detection framework, in: Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on, IEEE, 2002, pp. 202–215.
39. F. Cuppens, Managing alerts in a multi-intrusion detection environment, in: acsac, Vol. 1, 2001, p. 22.
40. A. Valdes, K. Skinner, Probabilistic alert correlation, in: Recent Advances in Intrusion Detection, Springer, 2001, pp. 54–68.
41. J. Sun, S. Papadimitriou, C. Faloutsos, Distributed pattern discovery in multiple streams, in: Advances in Knowledge Discovery and Data Mining, Springer, 2006, pp. 713–718.
42. J. Sun, C. E. Tsourakakis, E. Hoke, C. Faloutsos, T. Eliassi-Rad, Two heads better than one: pattern discovery in time-evolving multi-aspect data, Data Mining and Knowledge Discovery 17 (1) (2008) 111–128.
43. J. Sun, Incremental pattern discovery on streams, graphs and tensors, ACM SIGKDD Explorations Newsletter 10 (2) (2008) 28–29.
44. R. Yusof, S. R. Selamat, S. Sahib, Intrusion alert correlation technique analysis for heterogeneous log, IJCSNS International Journal of Computer Science and Network Security 8 (9) (2008) 132–138.
45. H. Debar, A. Wespi, Aggregation and correlation of intrusion-detection alerts, in: Recent Advances in Intrusion Detection, 2001, pp. 85–103.
46. M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.
47. C. Berge, Hypergraphs: Combinatorics of Finite Sets, North-Holland, 1989.
48. G. Vigna, A topological characterization of tcp/ip security, in: FME, 2003, pp. 914–939.
49. B. Morin, L. Mé, H. Debar, M. Ducassé, M2d2: A formal data model for ids alert correlation, in: RAID, 2002, pp. 115–127.

50. F. Baiardi, S. Suin, C. Telmon, M. Pioli, Assessing the risk of an information infrastructure through security dependencies, in: CRITIS, 2006, pp. 42–54.
51. W. Pieters, Ankh: Information threat analysis with actor-network hypergraphs, CTIT technical report series, Centre for Telematics and Information Technology, University of Twente, Enschede, 2010.
52. C. R. Johnson, M. Montanari, R. H. Campbell, Automatic management of logging infrastructure, in: National Centers of Academic Excellence - Workshop on Insider Threat, St Louis, MO, USA, 2010.
53. M. Korff, L. Ribeiro, Formal relationship between graph grammars and petri nets, in: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (Eds.), Graph Grammars and Their Application to Computer Science, Vol. 1073 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1996, pp. 288–303.
54. P. Alimonti, E. Feuerstein, Petri nets, hypergraphs and conflicts (preliminary version), in: E. W. Mayr (Ed.), Graph-Theoretic Concepts in Computer Science, Vol. 657 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1993, pp. 293–309.
55. A. Basu, R. W. Blanning, Metagraphs in workflow support systems, Decision Support Systems (3) 199 – 208.
56. A. Basu, R. W. Blanning, A formal approach to workflow analysis, Information Systems Research 11 (1) (2000) 17–36.
57. A. Basu, R. W. Blanning, Workflow analysis using attributed metagraphs, in: HICSS, 2001.
58. A. Basu, R. W. Blanning, Metagraphs and Their Applications, Integrated Series in Information Systems, Springer, Dordrecht, 2007.
59. A. Basu, R. W. Blanning, Metagraphs: a tool for modeling decision support systems, Manage. Sci. 40 (12) (1994) 1579–1600.
60. A. Polyvyanyy, M. Weske, Hypergraph-based modeling of ad-hoc business processes, in: Business Process Management Workshops, 2008, pp. 278–289.
61. P. Ammann, D. Wijesekera, S. Kaushik, Scalable, graph-based network vulnerability analysis, in: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002), Washington, DC, USA, 2002, pp. 217–224.
62. S. Noel, E. Robertson, S. Jajodia, Correlating intrusion events and building attack scenarios through attack graph distances, in: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004), Tucson, AZ, USA, 2004, pp. 350–359.
63. L. Wang, A. Liu, S. Jajodia, An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts, in: S. De Capitani di Vimercati, P. Syverson, D. Gollmann (Eds.), Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005), Vol. 3679 of Lecture Notes in Computer Science, Springer, Milan, Italy, 2005, pp. 247–266.
64. L. Wang, S. Noel, S. Jajodia, Minimum-cost network hardening using attack graphs, Computer Communications 29 (18) (2006) 3812–3824.
65. Y. Chen, B. W. Boehm, L. Sheppard, Value driven security threat modeling based on attack path analysis, in: HICSS, 2007, p. 280.
66. G. K. Palshikar, M. M. Apte, Collusion set detection using graph clustering, Data Min. Knowl. Discov. 16 (2) (2008) 135–164.
67. S. Bordoni, G. Facchinetti, Insurance fraud evaluation - a fuzzy expert system, in: FUZZ-IEEE - Proceedings of the 10th IEEE International Conference on Fuzzy Systems, IEEE, 2001, pp. 1491–1494.

68. D. Zhang, D. Gatica-Perez, S. Bengio, I. McCowan, Semi-supervised adapted hmms for unusual event detection, in: CVPR 2005 - IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 1, IEEE Computer Society, 2005, pp. 611–618.
69. J. Kim, K. Grauman, Observe locally, infer globally: A space-time mrf for detecting abnormal activities with incremental updates, in: CVPR 2009 - IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 2921–2928.
70. J. Yin, Q. Yang, J. J. Pan, Sensor-based abnormal human-activity detection, IEEE Trans. Knowl. Data Eng. 20 (8) (2008) 1082–1090.
71. D. H. Hu, X.-X. Zhang, J. Yin, V. W. Zheng, Q. Yang, Abnormal activity recognition based on hdp-hmm models, in: C. Boutilier (Ed.), IJCAI 2009 - Proceedings of the 21st International Joint Conference on Artificial Intelligence, 2009, pp. 1715–1720.
72. X.-X. Zhang, H. Liu, Y. Gao, D. H. Hu, Detecting abnormal events via hierarchical dirichlet processes, in: T. Theeramunkong, B. Kijsirikul, N. Cercone, T. B. Ho (Eds.), PAKDD 2009 - Advances in Knowledge Discovery and Data Mining, 13th Pacific-Asia Conference, Vol. 5476 of Lecture Notes in Computer Science, Springer, 2009, pp. 278–289.
73. F. Jiang, Y. Wu, A. K. Katsaggelos, Detecting contextual anomalies of crowd motion in surveillance video, in: ICIP 2009 - Proceedings of the International Conference on Image Processing, IEEE, 2009, pp. 1117–1120.
74. D. Mahajan, N. Kwatra, S. Jain, P. Kalra, S. Banerjee, A framework for activity recognition and detection of unusual activities, in: B. Chanda, S. Chandran, L. S. Davis (Eds.), ICVGIP 2004 - Proceedings of the Fourth Indian Conference on Computer Vision, Graphics & Image Processing, Allied Publishers Private Limited, 2004, pp. 15–21.
75. A. Mecocci, M. Pannozzo, A completely autonomous system that learns anomalous movements in advanced videosurveillance applications, in: ICIP 2005 - Proceedings of the 2005 International Conference on Image Processing, Vol. 2, IEEE, 2005, pp. 586–589.
76. F. Jiang, J. Yuan, S. A. Tsaftaris, A. K. Katsaggelos, Video anomaly detection in spatiotemporal context, in: ICIP 2010 - Proceedings of the International Conference on Image Processing, IEEE, 2010, pp. 705–708.
77. J. Wang, Z. Cheng, M. Zhang, Y. Zhou, L. Jing, Design of a situation-aware system for abnormal activity detection of elderly people, in: R. Huang, A. A. Ghorbani, G. Pasi, T. Yamaguchi, N. Y. Yen, B. Jin (Eds.), AMT 2012 - Active Media Technology, 8th International Conference, Vol. 7669 of Lecture Notes in Computer Science, Springer, 2012, pp. 561–571.
78. M. Albanese, C. Molinaro, F. Persia, A. Picariello, V. S. Subrahmanian, Finding "unexplained" activities in video, in: T. Walsh (Ed.), IJCAI 2011 - Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI/AAAI, 2011, pp. 1628–1634.
79. N. Vaswani, A. K. R. Chowdhury, R. Chellappa, "shape activity": A continuous-state hmm for moving/deforming shapes with application to abnormal activity detection, IEEE Transactions on Image Processing 14 (10) (2005) 1603–1616.
80. N. P. Cuntoor, B. Yegnanarayana, R. Chellappa, Activity modeling using event probability sequences, IEEE Transactions on Image Processing 17 (4) (2008) 594–607.

81. M. Brand, N. Oliver, A. Pentland, Coupled hidden markov models for complex action recognition, in: CVPR, IEEE Computer Society, 1997, pp. 994–999.
82. N. Oliver, E. Horvitz, A. Garg, Layered representations for human activity recognition, in: ICMI 2002 - 4th IEEE International Conference on Multimodal Interfaces, IEEE Computer Society, 2002, pp. 3–8.
83. R. Hamid, Y. Huang, I. Essa, Argmode-activity recognition using graphical models, in: CVPRW'03 - Conference on Computer Vision and Pattern Recognition Workshop, 2003, Vol. 4, IEEE, 2003, pp. 38–38.
84. M. Albanese, V. Moscato, A. Picariello, V. S. Subrahmanian, O. Udrea, Detecting stochastically scheduled activities in video, in: M. M. Veloso (Ed.), IJCAI 2007 - Proceedings of the 20th International Joint Conference on Artificial Intelligence, 2007, pp. 1802–1807.
85. S. Hongeng, R. Nevatia, F. Brémond, Video-based event recognition: activity representation and probabilistic recognition methods, Computer Vision and Image Understanding 96 (2) (2004) 129–162.
86. R. Xu, D. C. Wunsch, Survey of clustering algorithms, IEEE Transactions on Neural Networks 16 (3) (2005) 645–678.
87. T. Xiang, S. Gong, Video behavior profiling for anomaly detection, IEEE Trans. Pattern Anal. Mach. Intell. 30 (5) (2008) 893–908.
88. H. Zhong, J. Shi, M. Visontai, Detecting unusual activity in video, in: CVPR 2004 - IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 4, 2004, pp. 819–826.
89. C. E. Au, S. Skaff, J. J. Clark, Anomaly detection for video surveillance applications, in: ICPR 2006- 18th International Conference on Pattern Recognition, Vol. 4, IEEE Computer Society, 2006, pp. 888–891.
90. Y. Zhou, S. Yan, T. S. Huang, Detecting anomaly in videos from trajectory similarity analysis, in: ICME 2007 - Proceedings of the 2007 IEEE International Conference on Multimedia and Expo, IEEE, 2007, pp. 1087–1090.
91. L. Brun, A. Saggese, M. Vento, A clustering algorithm of trajectories for behaviour understanding based on string kernels, in: SITIS 2012 - Eighth International Conference on Signal Image Technology and Internet Based Systems, IEEE, 2012, pp. 267–274.
92. A. Adam, E. Rivlin, I. Shimshoni, D. Reinitz, Robust real-time unusual event detection using multiple fixed-location monitors, IEEE Trans. Pattern Anal. Mach. Intell. 30 (3) (2008) 555–560.
93. B. Mukherjee, L. T. Heberlein, K. N. Levitt, Network intrusion detection, Network, IEEE 8 (3) (1994) 26–41.
94. A. Jones, S. Li, Temporal signatures for intrusion detection, in: ACSAC 2001 - 17th Annual Computer Security Applications Conference, IEEE Computer Society, 2001, pp. 252–261.
95. P. Ning, Y. Cui, D. S. Reeves, Constructing attack scenarios through correlation of intrusion alerts, in: V. Atluri (Ed.), CCS 2002 - Proceedings of the 9th ACM Conference on Computer and Communications Security, ACM, 2002, pp. 245–254.
96. S. O. Al-Mamory, H. Zhang, Ids alerts correlation using grammar-based approach, Journal in Computer Virology 5 (4) (2009) 271–282.
97. X. Qin, W. Lee, Statistical causality analysis of infosec alert data, in: G. Vigna, E. Jonsson, C. Krügel (Eds.), RAID 2003 - Recent Advances in Intrusion Detection, 6th International Symposium, Vol. 2820 of Lecture Notes in Computer Science, Springer, 2003, pp. 73–93.

98. X. Qin, A probabilistic-based framework for INFOSEC alert correlation, Phd thesis, Georgia Institute of Technology (2005).
99. A. J. Oliner, A. V. Kulkarni, A. Aiken, Community epidemic detection using time-correlated anomalies, in: S. Jha, R. Sommer, C. Kreibich (Eds.), RAID 2010 - Recent Advances in Intrusion Detection, 13th International Symposium, Vol. 6307 of Lecture Notes in Computer Science, Springer, 2010, pp. 360–381.
100. P. Lancaster, K. Salkauskas, Curve and surface fitting. an introduction, London: Academic Press, 1986 1.
101. D. R. Karger, C. Stein, A new approach to the minimum cut problem, Vol. 43, 1996, pp. 601–640.
102. P. Mell, T. Bergeron, D. Henning, Creating a patch and vulnerability management program, NIST Special Publication 800-40, Version 2.0.
103. F. Foret, How to create and deploy a successful patch management policy and program, SANS Institute.
104. P. Mell, K. Scarfone, S. Romanosky, Common vulnerability scoring system, IEEE Security & Privacy 4 (6) (2006) 85–89.
105. The MITRE Corporation, Common Weakness Scoring System (CWSS™), http://cwe.mitre.org/cwss/, version 0.8 (June 2011).
106. Tenable Network Security®, The Nessus® vulnerability scanner, http://www.tenable.com/products/nessus (2014).
107. M. Dacier, Towards quantitative evaluation of computer security, Ph.D. thesis, Institut National Polytechnique de Toulouse (1994).
108. D. Zerkle, K. Levitt, NetKuang - A multi-host configuration vulnerability checker, in: Proceedings of the 6th USENIX Security Symposium, San Jose, CA, USA, 1996.
109. C. Phillips, L. P. Swiler, A graph-based system for network-vulnerability analysis, in: Proceedings of the New Security Paradigms Workshop (NSPW 1998), Charlottesville, VA, USA, 1998, pp. 71–79.
110. R. Ortalo, Y. Deswarte, M. Kaâniche, Experimenting with quantitative evaluation tools for monitoring operational security, IEEE Transactions on Software Engineering 25 (5) (1999) 633–650.
111. L. P. Swiler, C. Phillips, D. Ellis, S. Chakerian, Computer-attack graph generation tool, in: Proceedings of the DARPA Information Survivability Conference & Exposition II (DISCEX 2001), Vol. 2, Anaheim, CA, USA, 2001, pp. 307–321.
112. C. R. Ramakrishnan, R. Sekar, Model-based analysis of configuration vulnerabilities, Journal of Computer Security 10 (1/2) (2002) 189–209.
113. S. Jajodia, S. Noel, B. O'Berry, Managing Cyber Threats: Issues, Approaches, and Challenges, Vol. 5 of Massive Computing, Springer, 2005, Ch. Topological Analysis of Network Attack Vulnerability, pp. 247–266.
114. R. W. Ritchey, P. Ammann, Using model checking to analyze network vulnerabilities, in: Proceedings of the 2000 IEEE Symposium on Research on Security and Privacy (S&P 2000), Berkeley, CA, USA, 2000, pp. 156–165.
115. S. Jha, O. Sheyner, J. Wing, Two formal analyses of attack graphs, in: Proceedings of 15th IEEE Computer Security Foundations Workshop (CSFW 2002), Cape Breton, Canada, 2002.
116. M. Albanese, S. Jajodia, A. Singhal, L. Wang, An efficient approach to assessing the risk of zero-day vulnerabilities, in: Proceedings of the 10th International Conference on Security and Cryptpgraphy (SECRYPT), Reykjavik, Iceland, 2013.

117. T. Basar, The gaussian test channel with an intelligent jammer, IEEE Trans. Inf. Theor. 29 (1) (2006) 152–157.
118. A. Kashyap, T. Basar, R. Srikant, Correlated jamming on mimo gaussian fading channels, IEEE Transactions on Information Theory 50 (9) (2004) 2119–2123.
119. E. Altman, K. Avrachenkov, A. Gamaev, Jamming in wireless networks: The case of several jammers, in: Proceedings of the First ICST International Conference on Game Theory for Networks, GameNets'09, IEEE Press, 2009, pp. 585–592.
120. Q. Zhu, H. Li, Z. Han, T. Basar, A stochastic game model for jamming in multi-channel cognitive radio systems, in: ICC, IEEE, 2010, pp. 1–6.
121. Z. Han, N. Marina, M. Debbah, A. Hjørungnes, Physical layer security game: How to date a girl with her boyfriend on the same table, in: Proceedings of the First ICST International Conference on Game Theory for Networks, GameNets'09, IEEE Press, Piscataway, NJ, USA, 2009, pp. 287–294.
122. H. von Stackelberg, D. Bazin, R. Hill, L. Urch, Market Structure and Equilibrium, Springer, 2010.
123. T. Alpcan, S. Buchegger, Security games for vehicular networks, IEEE Transactions on Mobile Computing 10 (2) (2011) 280–290.
124. M. Raya, M. H. Manshaei, M. Félegyhazi, J.-P. Hubaux, Revocation games in ephemeral networks, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, 2008, pp. 199–210.
125. I. Bilogrevic, M. H. Manshaei, M. Raya, J.-P. Hubaux, Oren: Optimal revocations in ephemeral networks, Comput. Netw. 55 (5) (2011) 1168–1180.
126. Q. Zhu, L. Bushnell, T. Basar, Game-theoretic analysis of node capture and cloning attack with multiple attackers in wireless sensor networks., in: CDC, IEEE, 2012, pp. 3404–3411.
127. Q. Zhu, T. Basar, Dynamic policy-based ids configuration (2009) 8600–8605.
128. Q. Zhu, C. J. Fung, R. Boutaba, T. Basar, GUIDEX: A game-theoretic incentive-based mechanism for intrusion detection networks, IEEE Journal on Selected Areas in Communications 30 (11) (2012) 2220–2230.
129. M. H. Manshaei, Q. Zhu, T. Alpcan, T. Bacşar, J.-P. Hubaux, Game theory meets network security and privacy, ACM Comput. Surv. 45 (3) (2013) 25:1–25:39.
130. R. Dewri, N. Poolsappasit, I. Ray, D. Whitley, Optimal security hardening using multi-objective optimization on attack tree models of networks, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, ACM, New York, NY, USA, 2007, pp. 204–213.
131. R. Dewri, I. Ray, N. Poolsappasit, D. Whitley, Optimal security hardening on attack tree models of networks: a cost-benefit analysis, International Journal of Information Security 11 (3) (2012) 167–188.
132. N. Poolsappasit, R. Dewri, I. Ray, Dynamic security risk management using bayesian attack graphs, IEEE Trans. Dependable Secur. Comput. 9 (1) (2012) 61–74.
133. C. H. Papadimitriou, Computational complexity, Addison-Wesley, 1994.
134. D. S. Johnson, C. H. Papadimitriou, M. Yannakakis, On generating all maximal independent sets, Inf. Process. Lett. 27 (3) (1988) 119–123.
135. E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, Generating all maximal independent sets: Np-hardness and polynomial-time algorithms, SIAM J. Comput. 9 (3) (1980) 558–565.

136. A. Messac, A. Ismail-Yahaya, C. Mattson, The normalized normal constraint method for generating the pareto frontier, Structural and Multidisciplinary Optimization 25 (2) (2003) 86–98.
137. C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance, Branch-and-price: Column generation for solving huge integer programs, Operations Research 46 (3) (1998) pp. 316–329.
138. I. Muter, S. I. Birbil, K. Bülbül, Simultaneous column-and-row generation for large-scale linear programs with column-dependent-rows, Math. Program. 142 (1-2) (2013) 47–82.
139. S. Jajodia, S. Noel, P. Kalapa, M. Albanese, J. Williams, Cauldron: Mission-centric cyber situational awareness with defense in depth, in: Proceedings of the Military Communications Conference (MILCOM 2011), 2011.
140. Stanford Large Network Dataset Collection, Gnutella peer to peer network from august 4, 2002, http://snap.stanford.edu/data/p2p-Gnutella04.html (2014).
141. M. Ripeanu, A. Iamnitchi, I. T. Foster, Mapping the gnutella network, IEEE Internet Computing 6 (1) (2002) 50–57.