Selecting Countermeasures for ICT systems
Before They are Attacked
F. Baiardi, F. Tonelli, A. Bertolini, R. Bertolotti
Dipartimento di Informatica, Università di Pisa
Pisa, Italy
Email: {haruspex}@di.unipi.it

Abstract
A countermeasure is any change to a system to reduce the probability it is successfully attacked. We propose a model based approach that selects countermeasures through multiple simulations of the behaviors of an ICT system and of intelligent attackers that implement sequences of attacks. The simulations return information on the attacker sequences and the goals they reach we use to compute the statistics that drive the selection. Since attackers change their sequences as countermeasures are deployed, we have defined an iterative strategy where each iteration selects some countermeasures, updates the system models and runs the simulations to discover any new attacker sequence. The discovery of new sequences starts a new iteration.

The Haruspex suite automates the proposed approach. Some of its tools acquire information on the target system and on the attackers and build the corresponding models. Another tool simulates the attacks through the models of the system and of the attackers. The tool to select countermeasures invokes the other ones to discover how countermeasures influence the attackers. We apply the whole suite to three systems and discuss how the connection topology influences the countermeasures to adopt.

Keywords: Risk Assessment and Management; Countermeasures; Scenario; Monte Carlo Method.

1 Introduction
An intelligent attacker, or simply attacker, aims to acquire some access rights on an ICT system to exfiltrate or manipulate some information or to produce some unexpected behavior in a process the system controls. In general, the attacker collects these all these rights, its goal, through a sequence of attacks because one attack seldom grants all the rights of interest. An attacker selects the sequence to implement according to its priorities and preferences.

This paper introduces and applies a model-based approach to select countermeasures for an ICT system targeted by intelligent attackers. A countermeasure is any change to the system that reduces the success probability of one attack or guarantees its failure. The proposed approach introduces one model for the target system and one for each attacker. The system model describes attacks and their attributes such as the rights each attack grants and its success probability. An attacker model lists its goals and how it selects a sequence to these goals according to its priorities. The interaction between these models simulates the attacker behaviors. Since the output of each simulation strongly depends upon random events such as the success or the selection of attacks, we apply a Monte Carlo method and run independent simulations. Each simulation returns information on the attacker sequences, the goals they reach and the time this take. This defines a sample we use to compute the statistics to select countermeasures.

Haruspex [6–8] is a suite of tools to increase the robustness of a system before or after its deployment. Some tools build the system model and those of the attackers, other build a statistical sample through
Selecting Countermeasures for ICT systems Before They are Attacked

Baiardi et al.

an *experiment* that applies a Monte Carlo method with multiple simulations of a scenario where some attackers target the system.

Haruspex adopts a *divide et impera* strategy that only requires the probabilities of simple events such as the success of an attack. By collecting observations in multiple simulations, Haruspex tools return a statistical sample to compute global probabilities, such as the one that an attacker reaches a goal. The adoption of multiple simulations avoids the definition of a formal model that relates probabilities of simple events to the global ones. Being model based, the tools only require the models of the system and of the attackers and they can evaluate how countermeasures affect the probability that attackers reach their goals even before the system is actually deployed and attacked.

This paper is focused on the *manager*, the Haruspex tool to select countermeasures. This tool implements a first experiment to discover the attacker sequences. After selecting countermeasures for these sequences, it updates the system model and runs another experiment to discover the attacker sequences against the new system version. If the attackers still reach their goals, the *manager* selects further countermeasures, updates accordingly the system model, and it runs a new experiment till it discovers all the sequences.

After describing the *manager*, we present a case study that applies the whole suite to three systems and show how an increasing number of connections influences the countermeasures to deploy. The three systems do not actually exist but they merges features of real systems we have analyzed through the suite.

The paper is structured as follows. Sect. 2 briefly reviews related works on vulnerabilities, attacks, attacker, and attack simulation. Sect. 3 describes the building of the system model and of those of the attackers as well as the simulation of the attackers. Sect. 4 describes the *manager* and outlines how it iteratively selects countermeasures and evaluates their effectiveness. We compare the robustness of the alternative versions of a system the *manager* considers through the *security stress*. Sect. 5 defines this measure and then describes the case study. Lastly, Sect. 6 draws some conclusions and outlines some future works.

With respect to our previous works [6–9], this one is focused on the selection of countermeasures and it extends the one in the Parallel and Distributed Processing 2015 Conference [9] with a fully original case study to outline the relation between the selection of countermeasures and the connection topology. The original theoretical contribution of this work concerns the discussion of the problems posed by the adoption of countermeasures that reduce the success probability of attacks but do not guarantee their failure.

2 Related Works

We outline the contribution of the Haruspex suite by reviewing related works on attacks, plans, their description, and countermeasures.

[16, 20, 29, 32, 37, 39, 41] review the simulation of ICT attacks but do not adopt the Monte Carlo method. [17, 45] discuss intelligent, goal-oriented terrorists. The model of attack sequences in [18] is similar to the Haruspex one because it formally defines both pre and post condition of attacks but it does not discuss the probability of reaching a goal. [11] describes attack attributes and maps attacks into the proper countermeasures. [14, 23, 45] describe how the deployment of countermeasures affects the attackers. None of these work discusses attack sequences.

[12, 30] analyze attack simulation in the framework of game theory. In the same framework, [31] computes the best protection for alternative targets of an attacker. Instead, Haruspex focuses on an effective protection for a single target by reducing the probability that attackers reach their goals and not by diverting them to a distinct target [26, 27].

[2, 42] review agent-based simulation [21, 36, 43] consider multi objective optimization that under-
lies the selection of sequences. [28] discusses the relation between planning and attack sequences under
the assumption that accurate and complete information on the target system is available. [41] models at-
tacker with partial information and [40] defines a notion of look-ahead but in a different perspective than
Haruspex. Our attacker model is more general than those in these papers because we are not interested
in the optimal sequence or in the optimal strategy to select a sequence [50]. Instead, we focus on an
accurate modeling of how attackers:

- acquire information on the target system,
- select their sequence,
- change this selection because of countermeasures.

Most works on attack sequences do not discuss the selection of countermeasures. This is likely due
to the lack of formal models to compute the success probability of a sequence. The taxonomy of attacks
in [33] focuses on a series of security incidents. [24] proposes a classification to map each vulnerability
into a distinct class. The theoretical approach in [3] analyzes attack sequences targeting distinct network
nodes and it is focused on the compromised level of each node. This approach cannot discover all
the sequences because they grow exponentially in the number of attacks. [48] describes the discovery
of attack sequences and it computes the success probability of each one in isolation without considering
that distinct sequences may be selected. [5, 19, 34] model the selection of countermeasures through attack
graphs but they neglect the success probability of a sequence. [25] considers goal oriented attackers. [38]
discusses a metrics to evaluate system robustness. [1, 4] discuss the measurement of the risk due to,
respectively, ICT systems and software components.

Sequences of attacks also play a critical role in intrusion detection. [22, 44, 52, 53] correlate, attacks
or alerts from an intrusion detection system to discover attacks that belongs to the same sequence.

3 Haruspex Suite: Simulating a Scenario

The builder, the descriptor, and the engine are the suite tools that support the simulation of a scenario
where some attackers target a system $S$. The builder builds the model that describes the vulnerabilities in
the components of $S$ and the corresponding attacks. Instead, the descriptor receives information about
each attackers and it builds the corresponding model. Both tools minimize the complexity of model
building to increase not only the accuracy of the simulation but also the complexity of the scenarios that
can be analyzed. The engine uses the models built by the other tools to apply the Monte Carlo Method
and run an experiment with multiple simulations. In this way, it returns a sample to compute the statistics
to select and evaluate countermeasures. In the following, we use the acronyms in Table 1.

3.1 The Builder

The system model of $S$ is modular as it decomposes $S$ into some components, each defining some op-
erations. Through the attacks enabled by the vulnerabilities in a component $c$, an attacker can illegally
acquire some access rights, or rights, to invoke some operations. Vulnerabilities in $c$ are either known or
suspected. A known vulnerability is public when $S$ is analyzed. Instead, suspected vulnerabilities may
be discovered and become public in the future. Haruspex introduces these vulnerabilities to support a what-if
approach that evaluates how some vulnerabilities affect the selection of countermeasures. As an
example, we can evaluate how a stack overflow attack against $S$ influences the countermeasures to adopt.
We pair a suspected vulnerability with the probability it becomes public at each time $t$. 
Table 1: List of Components and Attributes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>the target system</td>
</tr>
<tr>
<td>$n$</td>
<td>a node of $S$</td>
</tr>
<tr>
<td>$c$</td>
<td>a component of $S$</td>
</tr>
<tr>
<td>$op$</td>
<td>an operation of a component</td>
</tr>
<tr>
<td>$ag$</td>
<td>an attacker</td>
</tr>
<tr>
<td>$g$</td>
<td>a goal of an attacker</td>
</tr>
<tr>
<td>$at$</td>
<td>an attack</td>
</tr>
<tr>
<td>$v$</td>
<td>a vulnerability</td>
</tr>
<tr>
<td>$n_a$</td>
<td>a node of an attack graph</td>
</tr>
<tr>
<td>$ar$</td>
<td>an arc of an attack graph</td>
</tr>
<tr>
<td>$r(n_a)$</td>
<td>the set of rights that $n_a$ represents</td>
</tr>
<tr>
<td>$v(at)$</td>
<td>the vulnerabilities enabling $at$</td>
</tr>
<tr>
<td>$time(at)$</td>
<td>the time to execute $at$</td>
</tr>
<tr>
<td>$pre(at)$</td>
<td>the rights to execute $at$</td>
</tr>
<tr>
<td>$post(at)$</td>
<td>the rights acquired if $at$ is successful</td>
</tr>
<tr>
<td>$succ(at)$</td>
<td>the success probability of $at$</td>
</tr>
<tr>
<td>$\lambda(ag)$</td>
<td>the look-ahead of $ag$</td>
</tr>
<tr>
<td>$na(ag)$</td>
<td>the number of attacks before a new selection</td>
</tr>
<tr>
<td>$sa$</td>
<td>a sequence of with $l$ attacks</td>
</tr>
<tr>
<td>$sa(i)$</td>
<td>the $i-th$ attack of $sa$, where $i \leq l$</td>
</tr>
<tr>
<td>$succ(ag, g, sa)$</td>
<td>the probability $sa$ enables $ag$ to reach $g$</td>
</tr>
<tr>
<td>$p(sa, g)$</td>
<td>the plan corresponding to $sa$</td>
</tr>
<tr>
<td>$cont(at)$</td>
<td>a countermeasure for $at$</td>
</tr>
<tr>
<td>$cost(at)$</td>
<td>the cost of $cont(at)$</td>
</tr>
<tr>
<td>$\text{lowrisk}$</td>
<td>an upper bound on the success probability of $ag$</td>
</tr>
</tbody>
</table>

Haruspex models an attacks $at$ through a set of attributes. $pre(at)$, the pre condition of $at$, includes the rights an attacker needs to implement $at$. $post(at)$, the post conditions of $at$, includes any rights an attacker acquires if $at$ is successful. $succ(at)$, the success probability of $at$, models the complexity of the actions of $at$ as well as the likelihood of events enabling its execution. As an example, $succ(at)$ is close to zero if the attacker has to execute the corresponding actions in a small time window that it does not control.

The builder receives a database with the vulnerabilities in the nodes and the interconnection structure of $S$. If $S$ already exists, this database is the output of a vulnerability scanning. The scanning of a node $n$ discovers the components that $n$ runs and returns a list of their public vulnerabilities. Also the interconnection structure components are scanned and their vulnerabilities added to the database.

If Haruspex is applied in the design of $S$, then the information in the database is deduced from public vulnerabilities in the components to be adopted.

The builder discovers attack attributes by classifying each vulnerability $v$ in its input database into one of seven classes. Vulnerabilities in the same class enable attacks with similar pre and post conditions. As an example, one class includes all the vulnerabilities of $n$ that only attackers with an account on $n$ can exploit. The vulnerabilities that enable attacks that do not require an account on $n$ belong to another class because these attacks have a distinct precondition. The classification of $v$ is driven by the description of the attacks that $v$ enables. We use the description in the Common Vulnerabilities and Exposures (CVE), a de-facto standard for vulnerability description. The builder analyzes the CVE description of $v$ and
selects its Common Vulnerability Scoring System score [49] to compute other attributes of the attacks \( v \) enables such as their success probabilities and execution time. We refer to [6, 10] for a description of the builder implementation and an evaluation of the accuracy of the classification.

The attack surface of \( n \) is an important attribute of the model of \( S \). This surfaces describes how an attacker sequence spreads among distinct nodes as it includes the attacks that other nodes of \( S \) can launch against \( n \). To compute this attribute, the builder integrates the information on the attacks enabled by the vulnerabilities of \( n \) with the topology of the logical connections to/from \( n \).

The builder stores the model of \( S \) in a database with the information previously described.

### 3.2 The Descriptor

An attacker \( ag \) owns the resources and the capability to violate the security policy of \( S \) to reach one of its goals. Each goal \( g \) of \( ag \) is a distinct sets of rights. To create the model of \( ag \) the user supplies to the descriptor information on the resources it can access to implement an attack and the operations \( ag \) is entitled to invoke by its initial rights. A further critical information is the strategy of \( ag \) to select a sequence of attacks according to its preferences and priorities [13]. We describe this strategy through \( AttGr(S,ag) \), the attack graph of \( ag \) against \( S \). \( AttGr(S,ag) \), is an oriented graph that represents all the sequences of \( ag \) to reach \( g \). Any node \( n_a \) of \( AttGr(S,ag) \) represents a set of rights \( r(n_a) \) and each arc \( ar \) is labeled by an attack \( at(ar) \). If \( ar \) is an arc from \( n_s \) to \( n_d \), then \( r(n_s) \) includes \( pre(at(ar)) \) and \( r(n_d) \) is the union of \( r(n_s) \) and of \( post(at(ar)) \). If \( r(n_i) \) is the initial set of rights of \( ag \) then \( n_i \) is the initial node of \( AttGr(S,ag) \). A path from \( n_i \) to any node \( n_f \) where \( r(n_f) \) is a goal of \( ag \) represents a sequence to reach the goal. Another notion of interest is the one of plan. A sequence of attacks to reach one of its goals is a plan if \( ag \) does reach the goal if it does not execute even one attack in the sequence.

If \( ag \) uses \( AttGr(S,ag) \) to select the sequence to implement, it always selects a plan, i.e. a sequence without useless attacks. However, this is too complex for any real system because the time to build \( AttGr(S,ag) \) is exponential in the size of \( S \). As a consequence, \( ag \) builds and analyzes \( subAttGr(S,ag,\lambda(ag),c) \) a small subset of \( AttGr(S,ag) \). \( c \), the initial node of the subset, is the current node of \( ag \), the one that describes the current rights of \( ag \). \( \lambda(ag) \), the other parameter that define the subset is a natural number, the look-ahead of \( ag \). If \( \lambda(ag) = 0 \), then \( subAttGr(S,ag,\lambda(ag),c) \) only includes \( c \) and the arcs leaving it. Here, \( ag \) randomly selects one of these arcs and the corresponding attack. If \( \lambda(ag) > 0 \), then \( subAttGr(S,ag,\lambda(ag),n) \) includes \( c \) and the paths of \( AttGr(S,ag) \) from \( c \) with at most, \( \lambda(ag) \) arcs. If at least one of these paths leads to a goal, then \( ag \) ranks all and only the sequences paired with a path to a goal. Otherwise, it ranks all the sequences paired with a path of \( subAttGr(S,ag,\lambda(ag),n) \). In both cases, the ranking considers the attributes of the attacks of each sequence. If no path in \( subAttGr(S,ag,\lambda(ag),n) \) leads to a goal, \( ag \) may select a sequence with useless attacks. Hence, \( ag \) reduces the complexity of selection by analyzing a subset of \( AttGr(S,ag) \) but, as a counterpart, it may select sequences with useless attacks.

To simulate in accurate way an attacker, we also consider how \( ag \) acquires the information to build \( subAttGr(S,ag,\lambda(ag),n) \) through a vulnerability scanning. This scanning returns all the vulnerabilities of the components in the scanned nodes and it delays \( ag \) for a time depending on these nodes. \( ag \) is delayed to scan a node \( n \) only the first time it ranks a sequence with an attack enabled by a vulnerability of a component running on \( n \). Hence, the collection overhead increases with the number of nodes \( ag \) scans for each selection that, in turn, increases with \( \lambda(ag) \). This is another compromise between accuracy and overhead of a selection. We model insiders by pairing each attacker with the nodes it does not need to scan because it already knows their vulnerabilities.

The Haruspex model of \( ag \) can specify distinct selection strategies according to the priorities of \( ag \). Among them:

1. random: returns any sequence with the same probability,
2. maxProb: returns the sequence with the best success probability,

3. maxIncr: returns the sequence granting the largest set of rights,

4. maxEff: returns the sequence with the best ratio between success probability and execution time.

None of these strategies neglects a sequence. As an example of a strategy that neglects a sequence, considers the one that never selects a sequence with an attack $a$ where $\text{succ}(at) \leq \beta$. We discuss in the following how this impacts the selection of countermeasures.

$ag$ invokes again its selection strategy after implementing $na(ag)$ attacks of the selected sequence. $na(ag)$ determines the compromise between the selection overhead and the ability of collecting more accurate information on $\text{AttGr}(S, ag)$ after some attacks. Furthermore, a low $na(ag)$ enables $ag$ to exploit suspected vulnerabilities as soon as they are discovered.

### 3.3 The Engine

Using the model of $S$ and those of the attackers in a scenario, the engine runs an experiment to analyze the scenario. An experiment includes a number of independent runs that simulate, for the same time interval, the discovery of suspected vulnerabilities and how each attacker selects and implements its sequence. Initially, the engine determines the attacks each attacker can implement according to the resource it can access. Then, at each time step of each run, first of all the engine determines the suspected vulnerabilities that are discovered. Then, it considers each attacker $ag$ that still has to reach at least one goal and it is idle or it has just executed an attack. After building $\text{subAttGr}(S, ag, \lambda(ag), n)$, the engine applies the selection strategy of $ag$. If $ag$ cannot select a sequence, then it is busy for the time to collect the information to build $\text{subAttGr}(S, ag, \lambda(ag), n)$ and then it waits for the discovery of a suspected vulnerability. If the strategy returns a sequence $sa$, the engine sequentially simulates the first $na(ag)$ attacks of $sa$ and $ag$ will be busy for the time to select $sa$ and the sum of times to successfully execute these attacks. The engine repeats a failed attack for an user-defined number of times before selecting a distinct sequence. Anytime an attack is successfully, the engine checks if $ag$ has reached a goal.

At the end of each run, the engine inserts into the output database one observation that records, among others, the sequence of each attacker, any goal it has reached, the time this has required. An observation also records information on $S$ such as the number of successful executions and failures of each attack. Before starting a new run, the engine restores the initial state of $S$ and of any attacker to guarantee run independence.

The observations in the database define a sample to compute statistics of interest. The number of runs in the experiment determines the confidence level of these statistics because each run returns one observation. The user can either choose the number of runs in an experiment or define the confidence level for some predefined statistics. In the latter case, engine starts a new run until reaching the required level.

The current version of the engine is coded in Java and it runs on a highly parallel IBM cluster with 96 cores. We exploit run independence to map distinct runs onto distinct cores. This results in a linear speed up.

### 4 The Haruspex Manager

This section describes how the manager selects countermeasures. For the sake of simplicity, we consider scenarios with just one attacker $ag$ with just one goal $g$. Generalization to multiple attackers with some goals are straightforward. We also assume that the user of the manager specifies lowrisk, the highest
probability it is willing to accept that ag reaches g. In the following, we do not discuss the implementation of \( \text{cont}(at) \), the countermeasure for \( at \), but only the decrease of \( \text{succ}(at) \) that it produces.

### 4.1 Sequences and Their Success Probabilities

Countermeasures should reduce \( \text{succ}(ag,g,sa) \), the probability \( ag \) reaches \( g \) through \( sa \), for any sequence \( sa \) that \( ag \) may implement to reach \( g \). \( \text{succ}(ag,g,sa) \) increases with the probability that \( ag \) selects \( sa \) as well as with the success probability of \( sa \). The former is related to the selection strategy of \( ag \), while the latter increases with the success probabilities of attacks in \( sa \). After running a Haruspex experiment, we approximate \( \text{succ}(ag,g,sa) \) as the percentage of runs where \( ag \) implements \( sa \) and reaches \( g \). We cannot approximate the probability that \( ag \) selects \( sa \) because \( ag \) may change its selection after some attack failures.

A countermeasure affects \( sa \) if it changes the success probability of at least one of its attacks. This change also affects the probability that \( ag \) selects \( sa \). Hence, the countermeasure may also change \( \text{succ}(ag,g,sqalt) \) where \( sqalt \neq sa \) by changing the probability that \( ag \) selects \( sqalt \). As an example, this may happen if \( ag \) adopts the \( \text{maxProb} \) strategy. We have experimentally verified that a countermeasure may force \( ag \) to select plans with a better success probability that it neglects before the countermeasure is deployed. This extends to ICT security the Braess’s paradox for traffic control \([15,51]\). While in traffic control the paradox is due to congestion, now \( ag \) neglects a plan with a better success probability because of partial information on \( S \) due to a low \( \lambda(ag) \). Since we do not know them in advance, we can discover all the sequences a countermeasure affect and their success probabilities by updating the model of \( S \) and by running an experiment with the new model.

These considerations have led to the design of an iterative algorithm where the \textit{manager} selects some countermeasures and runs a new experiment to discover any new sequences \( ag \) implements and their success probabilities. New iterations start till the overall success probability of \( ag \) is lower than \( \text{lowrisk} \). The update of the model of \( S \) exploits at best the Haruspex model based strategy to discover the effectiveness of countermeasure before their actual deployment.

### 4.2 Mapping Sequences into Plans

To select cost effective countermeasures, the \textit{manager} only deploys countermeasures for attacks that \( ag \) has to implement to reach \( g \). To this purpose, it applies the \textit{planner}, a tool that maps each sequence \( sa \) into the corresponding plan \( p(sa) \) to each sequence that \( ag \) implements to reach \( g \) in a run. We describe now how the \textit{planner} removes useless attacks through a backwards scans of \( sa \).

Initially, the \textit{planner} initializes \( tp(sa,g) \), the current approximation of \( p(sa) \), with the last attack of \( sa \). The \textit{planner} also initializes \textit{useful} to \( \text{pre}(sq(n)) \cup \{ g - \text{post}(sq(n)) \} \). This variable is a set with the rights that \( ag \) should own before executing the current attack to reach \( g \). Initially, this set includes the rights to execute \( sa(n) \) those in \( g \) that \( sa(n) \) does not grant.

The \textit{planner} does not add \( sa(j) \) to \( tp(sa,g) \) if and only if:

1. no right in \( \text{post}(sa(j)) \) belongs to \textit{useful},

2. before executing \( sa(j) \), \( ag \) already owns any right in \( \text{post}(sq(j)) \setminus \text{useful} \).

In 1), \( sa(j) \) is useless because no right it grants belongs to \( g \) or to the precondition of a useful attack. Instead, in 2) \( sa(j) \) is useless because it grants rights that \( ag \) owns initially or has already acquired through previous attacks.

If \( sa(j) \) is useful, before analyzing \( sa(j-1) \), the \textit{planner} removes from \textit{useful} the rights in the post condition of \( sa(j) \) and adds those in its pre condition.
At the end of the scanning, \( p(sa) = tp(sa,g) \).

This algorithm is correct provided that \( ag \) only executes attacks that grant some rights it does not own. A problem is arisen if \( sa \) interleaves more than one plan because the algorithm returns just one of these plans. We handle an interleaving by mapping any permutation of \( sa \) that is also a sequence, i.e. where the first \( j-1 \) attacks grants the rights in the pre condition of the \( j-th \) one.

After discovering any plan \( p \), the planner computes \( succ(ag,g,p) \) as the percentage of runs where \( ag \) reaches \( g \) through sequences mapped into \( p \).

4.3 Selecting Countermeasures for a Set of Plans

We assume that we know at least one countermeasure for each attack. If no countermeasure for \( at \) is known, then \( cost(at) \) is infinite. If some countermeasures for \( at \) are available, \( cont(at) \) is the one resulting in the largest reduction of \( succ(at) \). Ties are broken by selecting the cheapest one. As an example, if a patch for some vulnerability in \( vuln(at) \) is known, then \( cont(at) \) applies this patch and it guarantees the failure of \( at \). Here, \( cost(at) \) is the one of the patching. As an alternative, \( cont(at) \) may replace the component affect by \( vuln(at) \) with an equivalent one. An example where \( cont(at) \) only reduce \( succ(at) \) is the adoption of a longer encryption key or the adoption of an intrusion detection system.

The \( selector \) is the \( manager \) module that receives \( Sp \), a set of plans of \( ag \) to reach \( g \), and returns a set countermeasures to reduce the success probability of each plan in \( Sp \). To minimize the number of countermeasures, the \( selector \) considers the attacks the plans in \( Sp \) share because \( cont(at) \) affects all the plans that execute \( at \).

Initially, we assume that \( cont(at) \) guarantees the failure of \( at \). Then, we discuss the general case.

4.3.1 Zero Success Probability

The \( selector \) computes the countermeasures for \( Sp \) by considering the coverages of \( Sp \). A set of attacks is a coverage \([35]\) of \( Sp \) if the countermeasures for its attacks affects any plan in \( Sp \). The cost of a coverage is the sum of the costs of the countermeasures for its attacks.

The \( selector \) computes all the coverages for \( Sp \) and then returns the cheapest one. When computing a coverage, the \( selector \) neglects an attack \( at_1 \) if all the plans that share \( at_1 \) also share another attack \( at_2 \) such that \( cost(at_2) < cost(at_1) \). Furthermore, if some plans share more than one attack and their countermeasures have the same cost, the \( selector \) only considers the countermeasure resulting in the lowest success probability of the attack. The tool breaks further ties according to the ratio between the success probability of an attack and its execution time \([46–48]\).

The execution time of the \( selector \) is acceptable even if the coverage problem is NP-Complete provided that \( Sp \) is small or its plans shares a large number of attacks. Instead, the execution time sharply increases if \( Sp \) is large and the plans share a low number of attacks. However, if \( ag \) can implement a large number of plans, the deployment of countermeasures cannot result in a large robustness that requires an extensive redesign of \( S \).

If the \( selector \) returns a coverage with an infinite cost, then at least one plan only include attacks with no countermeasure. The success probability of this plan is a lower bound on the success probability of \( ag \).

4.3.2 Non Zero Success Probability

If \( cont(at) \) only reduces \( succ(at) \) for at least one attack \( at \), then we may have to select countermeasures for distinct attacks in the same plan. Obviously, this may reduce the success probability of \( ag \) only if sequences are not very short. As an example, a reduction of the success probability of each attack
in a sequence with two attacks can stop an attacker only if there are strong constrains on the time to reach a goal. Here, a set of attacks is a coverage if the countermeasures for its attacks reduces at least of $\delta$ the product of the success probabilities of the attacks in each plan. $\delta$ is a constant value that the selector determines according to the number of attacks in the plan. The selector prefers coverages with countermeasures that guarantee the failure of attacks.

In the following, we discuss further differences arising in this case.

### 4.4 Reducing the Success Probability of an Attacker

The manager runs a first experiment and it enters a loop. At first, each iteration applies the planner to the output of the previous experiment to discover each plan $p$ of $ag$ to reach $g$ and $\text{succ}(ag, g, p)$. Then, it invokes the selector to compute the countermeasures for these plans and it updates the model of $S$ to model the deployment. Then, the manager runs an experiment with the new model to discover any plan $ag$ successfully implements against the new version of $S$ and its success probability. $ag$ never executes these plans in a previous experiment because it selects them only when some countermeasures affect other plans. Only a new experiment can discover these plans because the simulation of the attacks of $ag$ against the previous versions of $S$ cannot return information to support their discovery. If, in the new experiment, the success probability of $ag$ is still larger than $\text{lowrisk}$, the manager starts a new iteration. Otherwise, it terminates after returning the countermeasures deployed in the last version.

The number of countermeasures the selector returns at each iteration strongly depends upon the plans it receives. In a global approach, at the $i$th iteration, the selector receives a set $Sp_i$ with the plans that $ag$ implements in any iteration. Hence, at each iteration the selector may return a set of countermeasures that is disjoint from those it has returned in the previous iterations. In the incremental approach, instead, $Sp_i$ only includes the plans $ag$ executes in the $i$th iteration. Then, the manager extends the countermeasures previously deployed with those the selector returns.

A global approach minimizes the number of countermeasures because it considers the attacks some plans share independently of the iteration that discovers a plan. Instead, the incremental approach cannot anticipate the plans $ag$ implements in the following iterations and the attacks they share with the previous ones. As a counterpart, this approach minimizes the number of plans that each iteration transmits to the selector.

The approach the manager adopts depends upon how countermeasures reduce the success probability of attacks.

#### 4.4.1 Zero Success Probability

If any $\text{cont}(at)$ guarantees the failure of $at$, the output of the selector in an iteration guarantees the failure of any plan, even if differs from the one of the previous iteration. Hence, $Sp_i$ includes all the plans the selector has received in the first $i-1$ iterations and a subset, $Cp_i$, of those $ag$ executes in the $i$th experiment. We insert plans into $Cp_i$ according to $\text{succ}(ag, g, p)$ and stop as soon as the sum of $\text{succ}(ag, g, p)$ for the remaining plans is lower than $\text{lowrisk}$. To reduce $Cp_i$ when $ag$ executes a large number of plans each with a low value of $\text{succ}(ag, g, p)$, we bound its size as a fixed percentage of successful plans in the $i$th iteration. This reduces the computational overhead of each iteration at the expense of the number of iterations because countemeasures may increase the success probability of some plans.

#### 4.4.2 Non Zero Success Probability

If some $\text{cont}(at)$ only reduce $\text{succ}(at)$, the selection of countermeasures for distinct attacks in a plan may affect in an unexpected way the plan success probability. This may results in a loop where the...
managers alternatively selects one of two sets of countermeasures. To avoid this loop, the manager adopts an incremental approach. Since the manager may transmit the same plan to the selector in distinct iterations, it also transmits the countermeasures previously selected.

4.5 Exiting the Iterations

The manager executes a finite number of iterations anytime any \( \text{cont}(at) \) results in the failure of \( at \) because each iteration discovers and stops at least one of finite number of plans. However, the number of iterations is unknown \textit{a priori} because the manager discovers each plan as \( ag \) implements it in an experiment. Since the success probability of \( ag \) decreases in a way that is not monotone, the user can bound the manager execution time by bounding the number of iterations. When the manager reaches this bound, it returns the best version of \( S \) it has discovered, i.e. the one with the lowest success probability of \( ag \).

If countermeasures only decrease the success probability of some attacks, then the incremental approach guarantees that the success probability of each plan steadily decreases because the number of countermeasures that affect a plan never decreases. However, since countermeasures may increase the success probability of \( ag \), even now a bound of the number of iterations may be specified.

4.6 Avoiding Iterations

The manager can adopt a distinct algorithm if \( ag \) never neglects a plan. Hence, in distinct manager experiments, \( ag \) selects distinct sequences till it executes all those it can select. In this way, it implements the same sequences of an attacker that randomly selects its sequence. This implies that we can discover all the sequences of \( ag \) through one experiment that adopts the \textit{random} selection strategy. Since no information is available on the sequence execution order, this solution may be adopted only if any \( \text{cont}(at) \) results in the failure of \( at \).

5 Case Study and Evaluation of Results

We have applied the Haruspex suite to for three ICT systems: \( syS_1 \), \( syS_2 \) and \( syS_3 \) with the same number of nodes but a different, complex, interconnection topology. Each ICT system has 24 subnetworks and a total of 36 distinct nodes running either Windows or Unix operating systems and provide a total of 175 services such as Telnet, SMB and Remote Desktop.

Fig.1, Fig.2 and Fig.3 show the three systems. Each figure shows the name of each subnet, the number of its nodes and the bidirectional, logical connections among subnets. The topology of \( syS_1 \) consists of 36 connections. \( syS_2 \) has four more connections that link, respectively, subnet \( E \) and subnet \( J \), \( D \) and \( O \), \( K \) and \( T \), and \( N \) and \( U \). \( syS_3 \) includes four further connections that link \( E \) and \( U \), \( D \) and \( T \), \( J \) and \( N \), and \( K \) and \( O \). There are 1848 vulnerabilities that affect the components of each system. The critical levels of these vulnerabilities ranges from critical to low.

Each system is the target of four attackers. Two attackers adopt the \textit{maxProb} strategy, the other two the \textit{maxIncr} one. The attackers with the same strategy have distinct \( \lambda \) values in the set \( \{1,2\} \). Each attacker initially controls the node in the \( A \) subnet and it aims to reach the control of, or implement a denial the service against, the node in the \( X \) subnet.

In the following, we compare the various versions of each system that the manager considers through the security stress. The security stress is a synthetic measure we have defined to evaluate and compare the robustness of alternative version of a system. As a particular case, we apply it to evaluate the effectiveness of the countermeasures the manager selects. First of all we describe this measure and then the selection of countermeasures for the three systems.
5.1 Security Stress

The security stress is a synthetic evaluation of how a system resists to some attackers. Initially, we consider a single attacker and generalizes at the end of this section. If $ag$ attacks $S$ to achieve $g$, $Str^S_{ag,g}(t)$, the security stress of $S$ at $t$, is the probability that $ag$ achieves $g$ within $t$. $Str^S_{ag,g}(t)$ evaluates the resistance of $S$ to the attacks of $ag$ as $t$ increases. $S$ cracks at $t_c$ if $ag$ always reaches $g$ for times larger than $t_c$. The resistance of $S$ in a time interval decreases as the surface underlying $Str^S_{ag,g}(t)$ increases.

To explain why we use $Str^S_{ag,g}(t)$ to evaluate the robustness of $S$, let us considers two time under the assumption they both exist. $t_0$ is the lowest time when the success probability of $ag$ is larger than zero while $t_1$ is the smallest time where this probability is 1. The value of $t_1 - t_0$ evaluates how long $S$, partially, resists to the attacks of $ag$ before cracking. The attacks are ineffective till $t_0$. Then, they are more and more effective till $S$ cracks at $t_1$ as $ag$ is always successful for larger times. The values of $t_0$
and of \( t_1 \) depend upon both some properties of \( S \), such as the attack attributes, and some properties of \( ag \), such as the sequences it selects. In particular:

1. \( t_0 \) depends upon the length of these sequences and the time the time to execute their attacks;

2. \( t_1 \) depends upon \( \text{succ}(at) \) that determines the average number of times \( ag \) repeats \( at \);

3. \( t_1 - t_0 \) depends upon the standard deviation of the length of the sequences of \( ag \).

We approximate \( \text{Str}^{S}_{ag,g}(t) \) as the percentage of the runs in an experiment where \( ag \) reaches \( g \) within \( t \). To evaluate the effectiveness of some countermeasures, we compare the robustness of \( S \) against the one of \( S_c \), the system that deploys the countermeasures. In general, \( \text{Str}^{S_c}_{ag,g}(t) \) is lower than \( \text{Str}^{S}_{ag,g}(t) \) in the time interval simulated in the Haruspex experiment. However, if some countermeasures forces \( ag \) to select shorter sequences to reach \( g \), \( \text{Str}^{S_c}_{ag,g}(t) \) may become larger than \( \text{Str}^{S}_{ag,g}(t) \) and the two curves may intersect.

If any scenario with multiple attackers, we consider the largest stress curve among those of the attackers and denote the corresponding attacker as the most dangerous one. If no curve is larger than the other, we consider a weighted sum of the curves.

An alternative definition of stress considers \( \text{Str}^{S}_{ag,g}(n) \), the success probability of \( ag \) after executing \( n \) attacks. This value includes both successful and failed executions and it evaluates both the effort of \( ag \) and the opportunities of \( S \) to detect the activity of \( ag \).

### 5.2 Selecting Countermeasures: A Case Study

Any Haruspex experiment described in the following has been implemented by the engine and it uses the models returned by the builder and by the descriptor. Each experiment reaches a 95\% confidence level on the components that are the targets of the attacker. This results in about 50,000 runs that our multicore architecture executes in about 10 minutes.

Fig.4 and Fig.5 show the stress curves of, respectively, \( sys_2 \) and \( sys_3 \) for each of the four attackers. We do not show the corresponding curves for \( sys_1 \) as they always overlap the x axis because no attacker reaches its goal.
Fig. 6 shows the stress of $sys_2$ due to the four attackers in terms of the number of attacks. This curve shows that the success probability of attacker is larger than zero after, at least, 14 attacks. Fig. 7 shows the corresponding stress for $sys_3$. When targeting $sys_3$, an attacker can reach its goal after, at least, 5 attacks, while the most powerful attacker always reaches its goals after 39 attacks.

The stress curves for $sys_2$ shows that there is not a most dangerous attacker. The stress curve of the one that adopts the $maxIncr$ strategy with $\lambda = 2$ dominates the other ones for most of the time but the first curve that reaches 1 is the one of the attacker with $maxProb$ strategy and $\lambda = 2$. In the following, we do not introduce a weighted sum of the curves of these two attackers and discuss each attacker independently.

The stress curve of the $maxIncr$ attacker with $\lambda = 2$ shows that $sys_2$ starts to crack after about 14 hours of attacks and it completely cracks after about 36 hours. Even the $maxProb$ attacker with $\lambda = 2$ starts to crack $sys_2$ after about 14 hours of attacks but it completely cracks $sys_2$ after about 30 hours. The least dangerous attacker against $sys_2$ adopts the $maxProb$ strategy with $\lambda = 1$. This attacker starts to crack the system after about 15 hours and it is always successful after 240 hours.
The stress curves of $sys_3$ show that most dangerous attacker adopts the $maxIncr$ strategy with $\lambda = 2$ and the least dangerous one is, for most of time, the one that adopts $maxProb$ with $\lambda = 1$. The most dangerous attacker starts to crack $sys_3$ only after 7 hours while its attacks are always successful provided that it has about 29 hours available. The least dangerous attacker completely cracks the system after 240 hours and starts to reach its goals after about 12 hours.

These curves show that a richer topology strongly may reduce the complexity of attacking a system. In fact, when passing from $sys_1$ to $sys_2$, the number of connections increases of about 11% and this enables any attackers to reach its goal. The same increase when passing from $sys_2$ to $sys_3$ reduces the shortest time to reach a goal from 14 to 7 hours.

We have applied the manager to compute the countermeasures for $sys_2$ and for $sys_3$. The manager adopts a global approach because countermeasures guarantee the failure of attacks. To avoid trivial solutions, we have assumed that there is no countermeasures for attacks that may be the first or the last ones of a sequence. These attacks are those the attacker can implement from the node it initially controls and those against the attacker goal, the subnet $X$ node.

With respect to $sys_2$, we have applied the manager to each of the two attackers with $\lambda = 2$ and that
adopt, respectively, maxProb and maxIncr. The manager returns for both attackers the same set with 7 countermeasures that guarantees that they cannot reach their goal. Three of these countermeasures concern the vulnerabilities on the SSH protocol in the nodes in the subnets M, T, and U. 3 further countermeasures patch 3 weakness of the Telnet, Samba and SSL protocols in the subnet M node. The last countermeasures changes a default password in subnet U node. By deploying countermeasures for less than 1% of vulnerabilities, the manager stops all the plans of these attackers but, since they never neglects a plan, the set of countermeasures also stops attackers that adopt distinct selection strategies.

The manager computes the countermeasures for the maxIncr attacker in 3 iterations while it computes the same set in 4 iterations for the maxProb attacker.

Fig. 8 and Fig. 9 show the stress values of the versions of sys2 that the manager considers in its iterations for the two attackers. The curve of the last version is not shown as it overlaps the x axis.

It is worth noticing that Fig. 9 shows an instance of the Braess’s paradox. The robustness of the version at the third iteration is lower than the one at the second iteration because the countermeasures that the manager selects in this iteration force the attacker to select longer sequences but with a better success probabilities. This increase in the success probability of the attacker is revealed by an intersection between the stress curves of the two versions. In this example, the attacker implements 412 distinct plans against the first version of sys2, 443 against the second version and 438 against the third one. As previously discussed, the attacker initially neglects some plans because of partial information on sys2 due to its λ.

The most dangerous attacker against sys3 adopts the maxIncr strategy with λ = 2. Fig. 10 shows the stress curves of the versions of sys3 that the manager produces. The manager selects the same set of countermeasures it computes to stop both attackers against sys2. These countermeasures are effective even for sys3 because they prevent the most dangerous attacker to exploit the new connections that sys3 offers. This shows that a richer topology may not increase the complexity of defending a system because the manager computes the countermeasures for sys3 in 3 iterations. However, this happens only when the manager exploits at best shared attacks among plans. As an example, this may not occur if an incremental approach is adopted. Even for sys3, the deployment of countermeasures for less than 1% of all the vulnerabilities stops all the attackers.

![Figure 8: sys2: Stress Curves at Distinct Iterations, maxIncr Attacker](image)

We have also consider the deployment of countermeasures that only reduce the success probability of attacks. As an example, by deploying a host intrusion detection system on each node that is the target of one plan of the attacker, we reduce the success probability of the most dangerous attacker against sys3 to 0.4 under the assumption that the false negative rate of the detection system is, at most, 1%. 

15
6 Conclusion

This paper has discussed how the Haruspex suite supports the selection of countermeasures for intelligent attackers. These attackers aim to reach some predefined goals by composing the attacks enabled by the system vulnerability into sequences. In Haruspex, two tools cooperate to discover an effective set of countermeasures: the manager and the planner. These tools implement an iterative process where each iteration implements a Haruspex experiment to discover how attackers change their sequences as countermeasures are deployed. This takes into account that an intelligent attacker can select and implement new sequences as old ones are affected by some countermeasures. We have presented a case study that applies the suite to select countermeasures for three systems where the complexity of interconnection topology increases. In this way, we have experimentally evaluated the influence of a richer topology on the complexity of countermeasure selection. Our experiments show some cases where the complexity of selecting a set of countermeasures does not increase with the number of connections provided that the selection considers the attacks that distinct plans share. We have also shown an example where the deployment of countermeasures increases the success probability of an attacker.
Future developments of the Haruspex suite concern the definition of more sophisticated models for the attackers and for the system and the modeling of computer worms.

References

[41] E. LeMay, W. Unkenholz, D. Parks, C. Muehrcke, K. Keefe, and W. H. Sanders. Adversary-driven state-


