Semi-automatically Augmenting Attack Trees using an Annotated Attack Tree Library

Ravi Jhawar[2], Karim Lounis[1], Sjouke Mauw[1,2], and Yunior Ramírez-Cruz[2]

[1]CSC, [2]SnT, University of Luxembourg
6, av. de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
{ravi.jhawar, karim.lounis, sjouke.mauw, yunior.ramirez}@uni.lu

Abstract. We present a method for assisting the semi-automatic creation of attack trees. Our method allows to explore a library of attack trees, select elements from this library that can be attached to an attack tree in construction, and determine how the attachment should be done. The process is supported by a predicate-based formal annotation of attack trees. To show the feasibility of our approach, we describe the process for automatically building a library of annotated attack trees from standard vulnerability descriptions in a publicly available online resource, using information extraction techniques. Then, we show how attack trees manually constructed from high level definitions of attack patterns can be augmented by attaching trees from this library.

Keywords: attack trees, semi-automatic construction, information extraction

1 Introduction

Attack trees are a well known graphical security model [19, 20, 15, 18, 11], widely used in industry and academia for modelling threats and conducting risk assessment [22]. In an attack tree, the root represents the goal of an attacker, which is refined into subgoals, represented by its children, each of which may be in turn subdivided into subgoals, and so on. Originally, two types of refinements were considered: disjunctive refinements, which represent the existence of several alternative ways of achieving a (sub)goal; and conjunctive refinements, which represent the need of jointly achieving a set of subgoals. Extensions of this model include the definition of new types of refinements, such as sequential conjunctions [16, 13, 24] and parallel disjunctions [12], as well as the possibility to insert countermeasures into the tree structure [11].

Due to their graphical nature, attack trees are a convenient tool for the description and presentation of attack scenarios in an easy-to-understand manner. Additionally, they have been equipped with formalisms for conducting quantitative risk analysis [4, 10, 7]. However, the model has not seen a widespread adoption in real-world risk analysis settings. The major factor behind this problem is the set of challenges faced when creating attack trees. Two main approaches have been followed for creating attack trees. They differ from each other in the balance between human effort and automation. On the one hand, a completely manual process [4, 22] is time- and labour-intensive, whereas on the other hand, completely automated approaches [23, 9, 8, 17, 5] tend to create trees that are too large, are hard to understand by humans, and have a hierarchical structure that does not fit the notion of refinement that analysts have, among other problems.

In this paper we explore one of the possible intermediate paths, where a part of the construction process is manually conducted by the experts and other parts are performed automatically. We view the construction of an attack tree as a semi-automatic process composed of the following steps:

1. A group of experts manually defines an initial version of the tree.
2. Some automated mechanism is used to enhance this tree using appropriate external resources.
3. The experts manually curate the new version of the tree.
4. Steps 2 and 3 are repeated until the final version of the tree is obtained.

The work presented in this paper focuses on the automatic component of this process, *i.e.* step number 2. Our goal is to provide a mechanism allowing the expert to profit from existing attack trees during the construction of a new one. To that end, we propose a method for augmenting an existing attack tree by attaching subtrees taken from a library, which is composed of existing attack trees. To support this process, we introduce an annotation of attack trees based on predicates representing the notions of assumptions and guarantees. This annotation is used to determine how the trees in the library can be attached to the tree in construction. The idea of formally annotating attack trees is not new. For example, Audinot et al. [1] use predicates representing pre-conditions and post-conditions on a state transition system to evaluate whether an attack tree is correct according to a system description. The purpose of our annotation is different. In our case, we do not have a system description, but a set of facts that are known to be true in the environment for which the tree is being constructed. More importantly, our annotation is not directly evaluated against a system model. Instead, we use it to decide whether, and how, a library tree can be used for augmenting the original attack tree.

Our contributions. This paper presents the following main results:

– We define a predicate-based annotation scheme for attack trees that allows to determine whether, and how, an attack tree can be attached to another as a subtree.
– We propose a method for augmenting an annotated attack tree by attaching annotated subtrees from a library.
– We develop an instantiation of our approach, whose contributions are two-fold:
  - We describe the process of automatically creating a large library of structurally simple annotated attack trees from a publicly available resource, the National Vulnerability Database[1] (NVD).

[1] `https://nvd.nist.gov/General`

- We demonstrate the use of this library to augment annotated attack trees manually constructed from standard attack pattern specifications, thus linking two existing resources, the NVD and the Common Attack Pattern Enumeration and Classification[2] (CAPEC).

The remainder of this paper is structured as follows. Section 2 introduces the main concepts used throughout the paper, along with our annotation scheme for attack trees. Then, Section 3 is devoted to the description of our method for augmenting an attack tree with trees from a library. Finally, we present the aforementioned instantiation of our method in Section 4 and provide our conclusions in Section 5.

2 Annotated attack trees

We will first introduce the terminology and definitions used throughout the paper. An attack tree is a rooted tree that graphically describes an attack scenario. The root of an attack tree represents the global goal of the attacker and the children of every node represent a set of subgoals, referred to as *refinements*, of the (sub)goal represented by that node. Finally, leaf nodes represent *basic actions* that can be executed by the attacker.

Let $\mathbb{B}$ represent the set of basic actions and let the symbols $\triangledown$ and $\triangle$ represent, respectively, disjunctive and conjunctive refinements. An attack tree is a closed term on the signature $\mathbb{B} \cup \{\triangledown, \triangle\}$, generated by the grammar

$$t ::= b | \triangledown(t, \ldots, t) | \triangle(t, \ldots, t) \tag{1}$$

where $b \in \mathbb{B}$. Figure 1 shows an example of an attack tree, which describes the CAPEC attack pattern 263, *Force Use of Corrupted File*[3], which describes an attack scenario where an application is forced to use a file that an attacker has corrupted, either by corrupting the sensitive file and waiting for the application to reload it, or by forcing the application to restart, thus reloading the file.

As we discussed previously, when manually constructing an attack tree, intuitive labels are often assigned to the nodes. While these labels may facilitate readability and visual analysis, they are informal, so it is not convenient to use them for automatically deciding whether a subgoal is susceptible of being refined, discriminate between several plausible refinements, etc. [1, 6]. To overcome this limitation, here we propose a formal annotation for the nodes of an attack tree. This annotation consists of pairs of predicates representing *assumptions* and *guarantees*. Assumptions encode properties assumed to hold in the contexts where the (sub)goal represented by each node is feasible, whereas guarantees are conditions that are ensured to hold if this (sub)goal is achieved. For example, in the tree shown in Figure 1, consider the subgoal labelled as *Force restart of application*. Intuitive examples of assumptions that this subgoal may be labelled

[2] `https://capec.mitre.org/index.html`
[3] `https://capec.mitre.org/data/definitions/263.html`

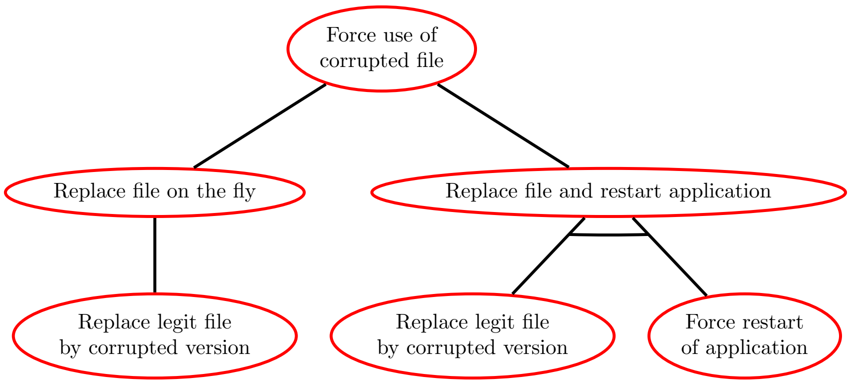

Force use of corrupted file
Replace file on the fly
Replace file and restart application
Replace legit file by corrupted version
Replace legit file by corrupted version
Force restart of application

Fig. 1. An attack tree representing CAPEC attack pattern 263.

with are those describing the operating system on which the target application runs. Likewise, an example of a guarantee for that subgoal is that the application is writing its output to a specific directory. While these examples are useful for illustrative purposes, it is clear that the nature of the assumptions and guarantees to use for the annotation needs to be expressible in a language that allows for effectively automatic processing. Examples of such languages are decidable subsets of first order logic. In Section 4, we will discuss a particular implementation of our approach that uses Prolog facts, automatically generated from publicly available information sources, as assumptions and guarantees.

In what follows, we will first formalise the notion of annotated attack tree, and then we will discuss the notion of consistency, which will be used for characterising valid annotations of attack trees. The consistency of annotations will play a key role in determining whether, and how, an attack tree can be attached to another one.

Let $\mathcal{E}$ be a set of predicates, or facts, and as above let $\mathbb{B}$ represent the set of basic actions. Additionally, we will use the symbol $\perp$ to represent a contradiction. An *annotated attack tree* is defined by the grammar

$$t ::= (p, q, b) \mid (p, q, \triangledown(t, \dots, t)) \mid (p, q, \triangle(t, \dots, t)) \tag{2}$$

where $p, q \in \mathcal{E}$ and $b \in \mathbb{B}$. In what follows, we will denote by $\mathcal{T}$ the set of all annotated attack trees. We define the functions $\varphi\colon \mathcal{T} \to \mathcal{E}$ and $\psi\colon \mathcal{T} \to \mathcal{E}$, which yield the assumptions and guarantees, respectively, associated to the root of an annotated attack tree, that is $\varphi(t) = p$ and $\psi(t) = q$.

Now, let $\Gamma \subseteq \mathcal{E}$. We say that Γ is *consistent* if $\Gamma \nvdash \perp$, that is, no contradictions are inferable from Γ. The purpose of Γ is to encode known properties of the environment in which the attacks described by the attack tree may take place. For example, when describing attacks on a computer system, the predicates in Γ may encode knowledge about the hardware and software, such as the operating system in use, the libraries and applications that have been installed, etc. Now consider, for example, that according to Γ the operating system is UNIX, and we have an attack tree t that, according to the assumptions $\varphi(t)$, describes attacks that exploit a vulnerability in a Windows library. Arguably, t does not represent a viable attack in this setting. An analogous reasoning is also valid for the guarantees. We also deem reasonable to discard the possibility of refining some subgoal of a tree with subtrees labelled with contradictory assumptions. Taking back the previous example, if a subgoal specifies the assumption that the operating system is UNIX, it makes no sense refining it with subtrees labelled with the assumption that the operating system is Windows. Additionally, we argue that, in a disjunctive refinement, the guarantees of a node should be inferable, in the context of Γ, from those of each subgoal; whereas the guarantees of a conjunctively refined node should be inferable from those of its subgoals combined. Finally, the guarantees for the subgoals of a conjunctively refined subgoal must not be in contradiction. We formalise the rationales discussed above as follows.

Definition 1. *Let t be an annotated attack tree and let Γ be a set of facts. We say that the annotation of t is* consistent *with Γ, and write* $\texttt{ConsAnnot}(t, \Gamma)$*, if the following conditions hold:*

$$\begin{aligned}
\texttt{ConsAnnot}((p, q, b), \Gamma) &= \Gamma, p, q \nvdash \bot && (3)\\
\texttt{ConsAnnot}((p, q, \bigtriangledown(t_1, \ldots, t_n)), \Gamma) &= \forall_{1\leq i\leq n}\, [\Gamma, p, \varphi(t_i) \nvdash \bot] \wedge && (4)\\
&\wedge\ \forall_{1\leq i\leq n}\, [\texttt{ConsAnnot}(t_i, \Gamma)] \wedge\\
&\wedge\ \forall_{1\leq i\leq n}\, [\Gamma, \psi(t_i) \vdash q]\\
\texttt{ConsAnnot}((p, q, \bigtriangleup(t_1, \ldots, t_n)), \Gamma) &= \Gamma, p, \varphi(t_1), \ldots, \varphi(t_n) \nvdash \bot\ \wedge && (5)\\
&\wedge\ \forall_{1\leq i\leq n}\, [\texttt{ConsAnnot}(t_i, \Gamma)] \wedge\\
&\wedge\ \Gamma, \psi(t_1), \ldots, \psi(t_n) \nvdash \bot\ \wedge\\
&\wedge\ \Gamma, \psi(t_1), \ldots, \psi(t_n) \vdash q
\end{aligned}$$

This definition of consistency will play a central role in our method for augmenting an attack tree using trees from a library. As we will discuss in the following section, our method will receive as inputs consistently annotated attack trees, and it will ensure that the augmented tree obtained as output continues to be consistently annotated.

3 Using a library of annotated attack trees for augmenting an annotated attack tree

As we discussed in the introduction, we view the construction of an attack tree as a semi-automatic process composed of four steps, out of which we focus on the second one, namely the use of an automated mechanism to augment a manually produced and/or curated version of an annotated attack tree, with the aid of appropriate external resources. Let t represent the current version of the annotated attack tree in construction. As we discussed in the previous section, the annotation of t is consistent with a given set of facts Γ. We define a *library of consistently annotated attack trees* as a set

$$\mathcal{L} \subseteq \{t' : t' \in \mathcal{T} \wedge t' \neq t \wedge \texttt{ConsAnnot}(t', \Gamma)\}. \tag{6}$$

In Section 4 we will describe in detail an example of how a large library of annotated attack trees can be constructed. Now, we will focus on describing how such a library can be used for augmenting an existing attack tree.

Given an annotated attack tree t an a library tree ℓ, the purpose of our method is to select a set of subtrees $s_1, s_2, \ldots, s_k$ in t, and add ℓ to the set of children of the roots of each s_i, thus obtaining a new tree t' identical to t, except for the fact that every (sub)goal represented by $s_1, s_2, \ldots, s_k$ contains ℓ as an additional subgoal. In these cases, we will say that ℓ has been *attached* to t as a subgoal of $s_1, s_2, \ldots, s_k$. For example, recall the attack tree depicted in Figure 1. If the library contains an attack tree describing how to remotely restart an application and another one describing how to presentially access a system and restart an application, both library trees can be attached to t as a disjunctive refinement of the leaf node labelled as *Force restart of application*.

The proposed method deals with two types of decisions. First, determining the subtrees of t to which library trees may be attached, and second, determining which library trees can be attached to those subtrees without violating the consistency of the annotation of the resulting tree.

Regarding the first issue, we consider two cases in which a library tree can be attached to a subtree s of t. The first case is that of an internal node which is disjunctively refined. In this case, we will allow a library tree to be added as an *additional* child of s. Doing this can be interpreted as providing a new alternative for achieving the subgoal represented by s. The second case is that of leaf nodes. In this case, we will allow a library tree to be attached as a *singleton disjunctive* refinement, so additional library trees can be attached afterwards. We deem it unnecessary to add subtrees to conjunctively refined internal nodes. The reason for this is the following. If a (sub)tree s of t has the form $(p, q, \triangle(t_1, \ldots, t_k))$, then from the fact that its annotation is consistent with Γ we have that $\Gamma, \psi(t_1), \ldots, \psi(t_k) \vdash q$. That is, the current subtrees already ensure all expected guarantees, so adding a tree ℓ will be redundant.

The second type of restriction limits the choices of library trees that can be attached to t. We will require that the annotation of the new tree t' obtained from t by attaching some library tree ℓ continues to be consistent with Γ. In order to avoid evaluating the consistency predicate every time we need to assess whether a library tree can be attached to t, we introduce the auxiliary predicate $\texttt{Attachable}(\ell, s, \Gamma)$ for a subtree s of t and a library tree $\ell \in \mathcal{L}$:

$$\texttt{Attachable}(\ell, s, \Gamma) = [\Gamma, \varphi(s), \varphi(\ell) \nvdash \bot] \wedge [\Gamma, \psi(\ell) \vdash \psi(s)] \tag{7}$$

The purpose of this predicate is to evaluate whether ℓ can be attached to t as part of a disjunctive refinement of s. As we will show next, if $\texttt{Attachable}(\ell, s, \Gamma)$ holds, then the tree obtained by attaching ℓ to t as a part of a disjunctive refinement of s continues to be consistently annotated with respect to Γ, which is the condition that we require.

Theorem 1. *Let $t, t' \in \mathcal{T}$ be two attack trees consistently annotated with respect to a set of facts Γ and let s be a subtree of t such that either $s = (p, q, \triangledown(t_1, \ldots, t_k))$ or $s = (p, q, b)$. Then, if $\texttt{Attachable}(t', s, \Gamma)$ holds, the annotation of the tree obtained from t by attaching t' as a subtree of s is also consistent with Γ.*

Proof. We first address the case where s is a disjunctively refined internal node of t. Let $s' = (p, q, \triangledown(t_1, \ldots, t_k, t_{k+1}))$, with $t_{k+1} = t'$, be the result of adding t' as a subtree of s in t; and let t'' be the annotated attack tree identical to t, except that s has been replaced by s'. From the consistency of the annotation of t, we have that

$$\forall_{1 \leq i \leq k} \left[\Gamma, p, \varphi(t_i) \nvdash \bot\right] \wedge \forall_{1 \leq i \leq k} \left[\texttt{ConsAnnot}(t_i, \Gamma)\right] \wedge \forall_{1 \leq i \leq k} \left[\Gamma, \psi(t_i) \vdash q\right].$$

Moreover, since we assume that $\texttt{Attachable}(t', s, \Gamma)$ holds, we have that

$$\left[\Gamma, p, \varphi(t_{k+1}) \nvdash \bot\right] \wedge \left[\Gamma, \psi(t_{k+1}) \vdash q\right].$$

Finally, according to the premises of the theorem, $\texttt{ConsAnnot}(t_{i+1}, \Gamma)$ holds, so we can conclude that

$$\forall_{1 \leq i \leq k+1} \left[\Gamma, p, \varphi(t_i) \nvdash \bot\right] \wedge \forall_{1 \leq i \leq k+1} \left[\texttt{ConsAnnot}(t_i, \Gamma)\right] \wedge \forall_{1 \leq i \leq k+1} \left[\Gamma, \psi(t_i) \vdash q\right].$$

Thus, $\texttt{ConsAnnot}(s', \Gamma)$ holds and, as a consequence, the new attack tree t'' is also consistently annotated with respect to Γ.

We now assume that s is a leaf node of t, and make $s' = (p, q, \triangledown(t'))$. Again, we have that $[\Gamma, p, \varphi(t') \nvdash \bot] \wedge [\Gamma, \psi(t') \vdash q]$ (because $\texttt{Attachable}(t', s, \Gamma)$ holds) and $\texttt{ConsAnnot}(t', \Gamma)$ holds because of the premises of the theorem. Consequently, we have that

$$\left[\Gamma, p, \varphi(t') \nvdash \bot\right] \ \wedge \texttt{ConsAnnot}(t', \Gamma) \ \wedge \ \left[\Gamma, \psi(t') \vdash q\right]$$

so $\texttt{ConsAnnot}(s', \Gamma)$ holds and, as a consequence, also does $\texttt{ConsAnnot}(t'', \Gamma)$. The proof is thus complete. □

With the previous definitions and results in mind, we now specify our tree augmentation strategy.

Definition 2. *Let t and ℓ be two attack trees consistently annotated with respect to a set of facts Γ. We define the function $r : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ as follows:*

$$r((p, q, b), \ell) = \begin{cases} (p, q, \triangledown(\ell)) & \textit{if } \texttt{Attachable}(\ell, (p, q, b), \Gamma) \\ (p, q, b) & \textit{otherwise} \end{cases} \tag{8}$$

$$r((p, q, \triangle(t_1, \ldots, t_n)), \ell) = (p, q, \triangle(r(t_1, \ell), \ldots, r(t_n, \ell))) \tag{9}$$

$$r((p,q,\bigtriangledown(t_1,\ldots,t_n)),\ell) =$$

$$= \begin{cases} (p,q,\bigtriangledown(t_1,\ldots,t_n,\ell)) & \textit{if } \forall_{1\le i\le n}\ [r(t_i,\ell)=t_i] \wedge \\ & \wedge\ \texttt{Attachable}(\ell,(p,q,\bigtriangledown(t_1,\ldots,t_n)),\Gamma) \\ (p,q,\bigtriangledown(r(t_1,\ell),\ldots,r(t_n,\ell))) & \textit{otherwise} \end{cases} \tag{10}$$

For an annotated attack tree t and an annotated library tree ℓ, $r(t,\ell)$ yields an annotated attack tree which is identical to t, except that ℓ has been attached to zero or more of its subtrees. From the definition of $r(t,\ell)$, note that when ℓ is attachable to a disjunctively refined node and also to some other node(s) in one or several of its subtrees, we only attach ℓ to the subtree(s), as far from the root as possible. This is simply a design choice, aiming to prevent an excessive growth of t, and doing the opposite would not be incorrect in terms of the consistency of the annotation of the resulting tree.

The function $r(t,\ell)$ can be efficiently computed by doing a post-order traversal of the structure of t. The time complexity of this operation is linear with respect to the number of nodes in t. Note, however, that its actual running time depends on that of the evaluation of the `Attachable` predicate, for which we will describe an efficient implementation in Section 4. As a final remark, note that the curator of the augmented tree may prefer to collapse singleton disjunctive refinements introduced by the our method, as this operation helps to reduce the size of the final tree. However, keeping the original tree as a subtree of the augmented one may be convenient in terms of readability, as we will see in the following section.

4 An instantiation of the proposed method using publicly available resources

In order to showcase the viability and usefulness of the proposed method, we first describe the process by which we created an annotated attack tree library from a publicly available database of vulnerability descriptions. Then, we discuss a case-study where this library is used for augmenting manually constructed annotated attack trees that describe well-known high level attack patterns.

We created the library from a subset of the entries of the NVD (National Vulnerability Database). This database contains a standardised repository of vulnerability descriptions, as defined by the CVE (Common Vulnerability and Exposures) List[4], enriched with meta-data such as platform information, severity scores, etc. NVD is publicly available[5] and its contents are frequently updated.

For our case-study, we used the releases of the database covering five years, from 2013 until the update corresponding to October 31st, 2017. This selection

[4] `http://cve.mitre.org/cve/`

[5] Available in `https://nvd.nist.gov/vuln/data-feed`. The data feeds are available in JSON and XML formats. For this case-study, we used the JSON releases.

contains 39,995 CVE's. We filtered the initial set of CVE's to discard those whose descriptions fail to comply with syntactic patterns that facilitate reliable automatic processing (as will be described in the next subsection) and obtained a final collection of 23,473 CVE's. From each of those, we created an entry in the annotated attack tree library. These entries can be seen in two manners. On the one hand, we can interpret each entry as a single-node tree specifying the action *Exploit the vulnerability described by CVE-YYYY-XXXX*. On the other hand, we can assume that further refinements of this action exist (or can be manually specified if needed) since the meta-data associated to CVE's contain a number of links to websites, some of which provide actual refinements in the form of detailed instructions, source code, etc. For the sake of simplicity, since our focus in this paper lies on the mechanisms to match library trees to subgoals of the attack tree being constructed, in our case-study we populated the library with single-node trees, annotated with assumptions and guarantees automatically extracted from the CVE's descriptions and meta-data.

Once the library was constructed, we manually defined high-level annotated attack trees that describe attack patterns from the Common Attack Pattern Enumeration and Classification (CAPEC), and automatically obtained augmented versions by attaching trees from the library.

In this instantiation of our method, and for the case study, we used Prolog (specifically SWI-Prolog[6]) to encode the assumptions and guarantees in the library trees. We also implemented in Prolog the rules to evaluate the predicate `Attachable`. A pipeline of Python scripts[7], along with the Stanford CoreNLP toolkit [14, 2], version 3.8.0[8], was used for the automatic library construction. Finally, the implementation of the function $r(t, \ell)$ was written in Python, and the freely available module PySWIP[9] was used to interact with the Prolog engine.

4.1 Automatic library construction

As we mentioned above, each tree in the library has the form $t_i = (p_i, q_i, b_i)$ and represents the action of exploiting a known vulnerability, as described by some CVE. We now discuss how the predicates for the assumptions p_i and the guarantees q_i are generated. The assumption predicates are a disjunction of facts of the form

$$\begin{array}{l}\texttt{affectedPlatform}(\texttt{cve}, [\texttt{vendor}, \texttt{product}, \texttt{version}])\\ \texttt{affectedPlatform}(\texttt{cve}, [\texttt{vendor}, \texttt{product}])\end{array}$$

extracted from the meta-data associated to the CVE, which uses the unambiguous Common Platform Enumeration[10] (CPE) naming scheme for systems,

[6] `http://www.swi-prolog.org/`

[7] The code and resources developed for our implementation are available at `https://github.com/yramirezc/lib-annotated-attack-trees`

[8] `https://stanfordnlp.github.io/CoreNLP/history.html`

[9] `https://github.com/yuce/pyswip`

[10] `https://nvd.nist.gov/Products/CPE`

applications, libraries, etc. For example, the metadata of NVD entry CVE-2017-6191 states that it affects the platform described as

$$\texttt{cpe}:2.3:\texttt{a}:\texttt{apng\_dis\_project}:\texttt{apng\_dis}:2.8:*:*:*:*:*:*:*.$$

From this metadata information, we generate the two following facts:

$$\texttt{affectedPlatform}(\texttt{cve\_2017\_6191}, [\texttt{apng\_dis\_project}, \texttt{apng\_dis}, 2.8])$$
$$\texttt{affectedPlatform}(\texttt{cve\_2017\_6191}, [\texttt{apng\_dis\_project}, \texttt{apng\_dis}]).$$

We now describe the generation of the guarantee predicates. Each guarantee is a conjunction of facts of the form

$$\texttt{allowedAction}(\texttt{cve}, [\texttt{s}, \texttt{v}, \texttt{o}])$$

where each triple $[\texttt{s}, \texttt{v}, \texttt{o}]$ represents an action. In order to obtain these triples, we used simple, yet highly reliable information extraction techniques, based on analysing the dependency trees of those sentences in the CVE descriptions whose syntactic structure matches some well-defined patterns.

The *dependency tree* of a sentence is a directed rooted tree that hierarchally organises (a subset of the) words according to their roles in the syntactic structure of the sentence. The edges of the dependency tree are called *dependencies*, and are labelled with information about the syntactic relation between the corresponding words. As an example, consider the following sentence, which is the description of the previously mentioned CVE-2017-6191:

Buffer overflow in APNGDis 2.8 and below allows a remote attacker to execute arbitrary code via a crafted filename.

Figure 2 shows the dependency tree of this sentence.

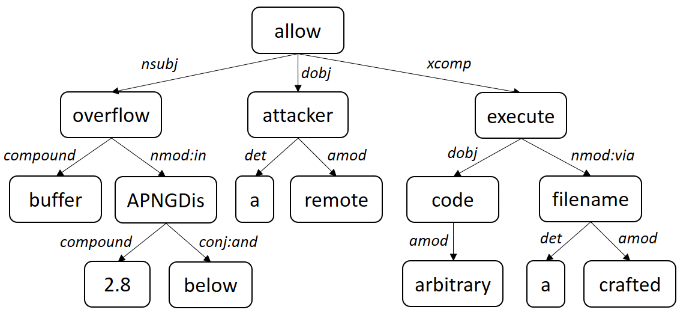
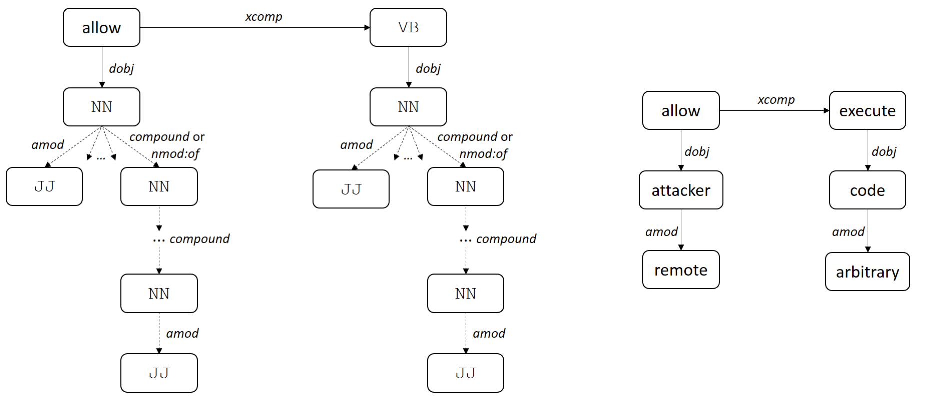


Fig. 2. Dependency tree of the description of CVE-2017-6191.

Several tools are available for performing dependency analysis. As we mentioned above, in this case-study we used the dependency parser module of the well-known Stanford CoreNLP toolkit [14, 2], version 3.8.0. The description of CVE-2017-6191 exemplifies the standardised language used in the subset of NVD entries that we selected for constructing the library. Even though these descriptions are given in natural language, they all contain at least one sentence whose dependency tree contains as a subtree one of the structures depicted in Figure 3 (a).



(a) Extraction pattern.

(b) Instance in description of CVE-2017-6191.

Fig. 3. Subtree-based extraction pattern used for obtaining guarantee predicates, and its instantiation in the description of CVE-2017-6191.

In the figure, solid arrows represent dependencies that must necessarily occur, whereas dashed arrows represent dependencies that may or may not occur. The labels `NN`, `VB` and `JJ` are the standard part-of-speech tags for nouns, verbs and adjectives, respectively; whereas the relevant dependency labels are *dobj*, *xcomp*, *amod*, *compound* and *nmod:of*. For a detailed description of the meaning of these labels and the linguistic foundation underlying them, we refer the reader to Appendix A and the works in [21, 3]. In subtrees with this structure, the direct object complement of the main verb (*to allow*) is the subject of the action represented by the open clausal complement (*xcomp*). Considering this, for every substructure of this type that we found, we generated a fact of the form

$$\texttt{allowedAction}(\texttt{cve}, [[w_1, \ldots, w_t], \texttt{vb}, [w'_1, \ldots, w'_{t'}]]),$$

where $[w_1, \ldots, w_t]$ contains the set of nouns and adjectives in the subject of `vb`, whereas $[w'_1, \ldots, w'_{t'}]$ contains the set of nouns and adjectives in the direct object complement. Following standard practices in information extraction, in all cases we collapsed nouns, verbs and adjectives to their canonical forms, or *lemmas*. As an example of the fact extraction process, recall the dependency tree of the description of CVE-2017-6191, shown in Figure 2. This tree contains as a subtree the structure depicted in Figure 3 (b), as a result of which we generate the fact

$$\texttt{allowedAction}(\texttt{cve\_2017\_6191}, [[\texttt{attacker}, \texttt{remote}], \texttt{execute}, [\texttt{code}, \texttt{arbitrary}]]).$$

4.2 Manual annotation of original trees and evaluation of the `Attachable` predicate

We now describe the manual annotation of assumptions and guarantees that must be conducted on the original trees so they can be augmented with trees from the library. To label assumptions, we used facts of the form

$$\texttt{assumedPlatforms}([\texttt{p}_1, \ldots, \texttt{p}_t]),$$

where each $\texttt{p}_i$ is a (possibly partial) platform specification. For its part, to label guarantees we used facts of the form

$$\texttt{requiredActions}([[\texttt{s}_1, \texttt{v}_1, \texttt{o}_1], \ldots, [\texttt{s}_t, \texttt{v}_t, \texttt{o}_t]]).$$

Aside from the manual annotation of any particular tree, we additionally specified a common set of Prolog rules for evaluating the predicate `Attachable` on any tree. By means of these rules, we make

$$\Gamma, \texttt{assumedPlatforms}([\texttt{p}_1, \ldots, \texttt{p}_t]), \texttt{affectedPlatform}(\texttt{cve}, \texttt{p}) \vdash \bot$$

if `p` matches no $\texttt{p}_i$, $1 \leq i \leq t$. In this case, `Attachable` evaluates to `false`; otherwise, the result depends on the evaluation of the assumptions. By `p` *matching* $\texttt{p}_i$, we mean that `p` refers to the same vendor as $\texttt{p}_i$, as well as to the same product and version, if $\texttt{p}_i$ specifies each of these pieces of information. If $\texttt{p}_i$ only specifies partial information, e.g. vendor only, and it coincides with the equivalent pieces of information in `p`, then `p` is also considered to match $\texttt{p}_i$. For example, [`cisco`, `residential_gateway_firmware`] matches itself, as well as [`cisco`]. For experimental purposes, we additionally allow the assumptions label of some nodes of the tree to be one of the facts `attachNothing`, which always makes `Attachable` evaluate to `false`; and `attachAnything`, which makes the final result depend on the evaluation of the guarantees. Finally, according to the defined rules, we make

$$\begin{aligned}&\Gamma, \texttt{allowedAction}(\texttt{cve}, [\texttt{s}_1, \texttt{v}_1, \texttt{o}_1]), \ldots, \texttt{allowedAction}(\texttt{cve}, [\texttt{s}_t, \texttt{v}_t, \texttt{o}_t]) \\ &\quad \vdash \texttt{requiredActions}([[\texttt{s}_1, \texttt{v}_1, \texttt{o}_1], \ldots, [\texttt{s}_{t'}, \texttt{v}_{t'}, \texttt{o}_{t'}]])\end{aligned}$$

if every required action $[\texttt{s}_i, \texttt{v}_i, \texttt{o}_i]$ matches an allowed action $[\texttt{s}_j, \texttt{v}_j, \texttt{o}_j]$. In this case, a match exists if $\texttt{v}_i$ and $\texttt{v}_j$ are identical, $\texttt{s}_i$ and $\texttt{s}_j$ satisfy set equality, and so do $\texttt{o}_i$ and $\texttt{o}_j$. For example, the action [[`remote`, `attacker`], `overwrite`, [`file`]] matches [[`attacker`, `remote`], `overwrite`, [`file`]], but does not match [[`attacker`], `overwrite`, [`file`]]. For experimental purposes, we additionally allow the guarantees label of some nodes of the tree to be the fact `everythingGuaranteed`, which makes the final result depend on the compliance of the library tree with the assumptions.

4.3 An example of the execution of our method

We now discuss in detail one example to show the characteristics of the augmented trees that can be obtained by our method. In order to manually create an initial attack tree, we selected from CAPEC the meta-attack pattern no. 165, *File Manipulation*, an instance of the category 262, *Manipulate System Resources*. Among the instances of this meta-attack pattern, we selected the attack pattern no. 263, *Force Use of Corrupted Files*, which has been discussed in previous sections (recall Figure 1 for reference).

We created the following four different annotated versions of this tree, which differ from each other in the assumptions used for labelling the leaf nodes:

i. All leaf nodes are labelled with `attachAnything`.
ii. All leaf nodes are labelled with `assumedPlatforms`([[`cisco`]]).
iii. All leaf nodes are labelled with `assumedPlatforms`([[`theforeman`, `foreman`]]).
iv. All leaf nodes are labelled with `assumedPlatforms`([[`theforeman`, `foreman`], [`cisco`]]).

For all four annotation variants, we specified the same guarantee annotation. Recall from Figure 1 that this attack tree has three leaf nodes, two of them labelled as *Replace legit file by corrupted version* and the other one labelled as *Force restart of application.* For simplicity, we will refer to the former two leaf nodes as b_1 and b_2, and to the latter as b_3. We labelled both b_1 and b_2 with the guarantee

`requiredActions`([[[`remote`, `attacker`], `overwrite`, [`arbitrary`, `file`]]]),

whereas b_3 was labelled with the guarantee

`requiredActions`([[[`remote`, `attacker`], `execute`, [`arbitrary`, `command`]]]).

Once the manual annotations were complete, we iteratively applied the function $r(t, \ell)$ on each annotated variant for every library tree. Table 1 summarises the number of library trees attached by applying the augmentation process to each annotated variant.

Variant	Attached to b_1	Attached to b_2	Attached to b_3	Total attached
i	10	10	182	202
ii	1	1	11	13
iii	1	1	2	4
iv	2	2	13	17

Table 1. Number of library trees attached to each annotated tree variant.

In every case, the leaves of the original annotated tree are now disjunctively refined with sets of library trees indicating which NVD vulnerabilities may be exploited to obtain the respective subgoals. As expected, for the first variant, which sets no *a priori* assumptions about the environment where the attack will be executed, the augmented tree is considerably large, which in turn makes it difficult to be read and used by human analysts. The remaining variants, by fine-tuning the platform-related assumptions, result in smaller augmented trees, which are not only easier to read and analyse by humans, but are arguably better suited to each specific scenario. To illustrate the output of our method, Figure 4 shows the augmented tree obtained for the annotated variant *iii*.

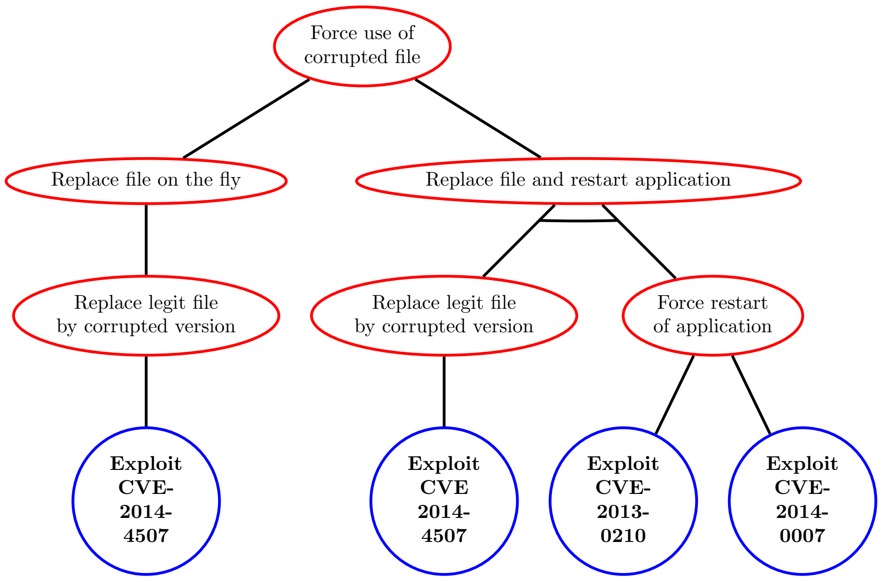

Force use of corrupted file
Replace file on the fly
Replace file and restart application
Replace legit file by corrupted version
Replace legit file by corrupted version
Force restart of application
Exploit CVE-2014-4507
Exploit CVE 2014-4507
Exploit CVE-2013-0210
Exploit CVE-2014-0007

Fig. 4. Augmented tree obtained for the annotated variant *iii*.

5 Conclusions and future work

In this paper we have presented a method for assisting the semi-automatic creation of attack trees by augmenting an attack tree with subtrees from a library. The process is supported by an annotation of attack trees based on assumption and guarantee predicates. We have showcased the feasibility of our approach by automatically generating a library of attack trees from standardised vulnerability descriptions in the NVD, and using trees from this library to augment a manually constructed annotated attack tree representing a high level attack pattern described in CAPEC.

The results presented in this paper can be extended in several interesting ways. For example, similar approaches may be useful for automatically generating libraries of countermeasures, which can in turn be used to augment attack-defence trees or to convert an attack tree into an attack-defence tree by semi-automatically attaching countermeasures. Moreover, in the instantiation of our method presented in Section 4, we can move from the current static library to a dynamic one, by exploiting the syndication services provided by NVD, which allow to access new and updated CVEs when they are available. A dynamic library would in turn allow analysts to maintain dynamic attack trees, which is an interesting research subject on its own.

Acknowledgements. The research reported in this paper received funding from Luxembourg's Fonds National de la Recherche (FNR), under grant C13/IS/5809105 (ADT2P).

References

1. M. Audinot, S. Pinchinat, and B. Kordy. Is my attack tree correct? In *European Symposium on Research in Computer Security*, pages 83–102. Springer, 2017.
2. D. Chen and C. Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.
3. M. C. De Marneffe and C. D. Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.
4. M. Ford, M. Fraile, O. Gadyatskaya, R. Kumar, M. Stoellinga, and R. Trujillo-Rasua. Using attack-defense trees to analyze threats and countermeasures in an ATM: A case study. In *Proc. of PoEM*. Springer, 2016.
5. O. Gadyatskaya. How to generate security cameras: Towards defence generation for socio-technical systems. *Lecture Notes in Computer Science*, 9390:50–65, 2015.
6. O. Gadyatskaya, R. Jhawar, S. Mauw, R. Trujillo-Rasua, and T. A. C. Willemse. Refinement-aware generation of attack trees. In *International Workshop on Security and Trust Management*, pages 164–179, 2017.
7. R. R. Hansen, P. G. Jensen, K. G. Larsen, A. Legay, and D. B. Poulsen. Quantitative evaluation of attack defense trees using stochastic timed automata. In *Proc. of GramSec 2017*, 2017.
8. J. B. Hong, D. S. Kim, and T. Takaoka. Scalable attack representation model using logic reduction techniques. In *Proc. of TrustCom*. IEEE, 2013.
9. M. G. Ivanova, C. W. Probst, R. R. Hansen, and F. Kammüller. Transforming graphical system models to graphical attack models. In *Proc. of GraMSec 2015*, 2015.
10. R. Jhawar, K. Lounis, and S. Mauw. A stochastic framework for quantitative analysis of attack-defense trees. *Lecture Notes in Computer Science*, 9871:138–153, 2016.
11. B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer. Attack-defense trees. *Oxford Univ. Press J. Logic and Computation*, 24(1):55–87, 2014.
12. K. Lounis. Stochastic-based semantics of attack-defense trees for security assessment. *Electronic Notes in Theoretical Computer Science*, 337:135–154, 2018.
13. W. P. Lu and W. M. Li. Space Based Information System Security Risk Evaluation Based on Improved Attack Trees. In *Procs. of MINES 2011*, pages 480–483, 2011.

14. C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.
15. S. Mauw and M. Oostdijk. Foundations of attack trees. In *International Conference on Information Security and Cryptology*, pages 186–198. Springer, 2005.
16. L. Piètre-Cambacédès and M. Bouissou. Beyond Attack Trees: Dynamic Security Modeling with Boolean Logic Driven Markov Processes (BDMP). In *Procs. of EDCC 2010*, pages 199–208, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
17. S. Pinchinat, M. Acher, and D. Vojtisek. ATSyRa: an integrated environment for synthesizing attack trees. In *Proc. of GraMSec 2015*, 2015.
18. A. Roy, D. S. Kim, and K. S. Trivedi. Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. *Security and Communication Networks*, 5(8):929–943, 2012.
19. B. Schneier. Attack Trees: Modeling Security Threats. *Dr. Dobb's Journal of Software Tools*, 24(12):21–29, 1999.
20. B. Schneier. *Secrets and lies: digital security in a networked world.* John Wiley & Sons, 2011.
21. S. Schuster and C. D. Manning. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Procs. of LREC 2016*, 2016.
22. A. Shostack. *Threat modeling: Designing for security.* Wiley, 2014.
23. R. Vigo, F. Nielsen, and H. R. Nielson. Automated generation of attack trees. In *Procs. of CSF 2014*, pages 337–350. IEEE, 2014.
24. J. Willemson and A. Jürgenson. Serial Model for Attack Tree Computations. In D. Lee and S. Hong, editors, *Procs. of ICISC 2009*, pages 118–128, 2010.

A List of dependency labels used in Section 4

These are the relevant dependency labels used in Subsection 4.1 for the extraction patterns that enable the automatic generation of guarantee predicates for the library trees:

– *dobj*: main noun of the direct object complement.
– *xcomp*: main verb of an open clausal complement. This is the relation between the main verb of the sentence (*to allow* in the cases processed here) and the main verb of a subordinate sentence serving as clausal complement (in this case, the subordinate sentence describing the action that is allowed).
– *amod*: adjectival noun modifier. This is the relation between the main noun of a noun phrase and an adjective that qualifies it, e.g *remote* and *attacker* or *arbitrary* and *code.*
– *compound*: noun compound modifier. Similar to the previous one, but referred to a composition of nouns, one of which modifies the other, e.g. the relation between *service* and *denial* in the noun phrase *service denial.*
– *nmod:of*: head of prepositional noun modifier introduced by the preposition *of.* Similar to the previous one, but referred to the composition of a noun and a prepositional phrase that modifies it, e.g. the relation between *service* and *denial* in the noun phrase *denial of service.*