Automated Attack Tree Generation Using Artificial Intelligence & Natural Language Processing

Alan Birchler De Allende[1], Bastien Sultan[1][0000−0002−5031−5794], and Ludovic Apvrille[1][0000−0002−1167−4639]

LTCI, Télécom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France
alan.birchler-de-allende@eurecom.fr, {bastien.sultan, ludovic.apvrille}@telecom-paris.fr

Abstract. Attack trees are widely used by engineers to analyze and document threats during system design. They are also particularly valuable for supporting the risk analysis of systems. Despite advancements in automation, the construction of these trees is often manual, leading to errors and the potential to overlook unconventional attack vectors. This paper introduces a novel method and tool that addresses a gap in existing literature: generating attack trees early in the design stage from a textual specification of the system, written in natural language. By leveraging natural language processing and large language models, this approach helps engineers identify threats concurrently with the initial design of the system, thus avoiding time-consuming re-engineering later on. Additionally, the paper introduces metrics to evaluate the syntactic and semantic correctness of the generated attack trees. Our contributions are assessed using attack trees generated from three different system specifications, with a comparative analysis based on the defined metrics between trees generated by the tool and those created by engineers. From these assessments, we have discovered that our methodology produces attack trees at a quality not that far off from that of an individual engineer.

Keywords: Artificial Intelligence, Attack Trees, Natural Language Processing

1 Introduction

Given the expanding attack surface of both information technology (IT) and embedded systems, the imperative to accurately identify potential threats and appropriate defensive measures has become increasingly critical. Attack trees, a prevalent methodology for modeling threats and vulnerabilities in security systems, offer security analysts a structured perspective for systematically identifying and mitigating cybersecurity risks [27]. Traditionally, the construction of attack trees, including attack-defense trees, has been a largely manual and labor-intensive process. It often requires substantial expertise and domain-specific knowledge to anticipate sophisticated or unconventional attack vectors.

Generative artificial intelligence (AI) is a promising avenue to address these challenges. With its capacity to work from large datasets, generative AI could potentially enhance domain-specific knowledge and improve coverage of possible attack scenarios.

This paper investigates the application of generative AI technologies, specifically large language models (LLMs), in the generation of attack trees. The root attack in an attack tree typically represents the primary objective of cybercriminals, such as stealing money or information, using a system as an attack relay, or causing catastrophic failure. Furthermore, sub-attacks that have a direct path to the root attack are supposed to demonstrate the steps that an attacker needs to achieve the root attack. A straightforward application might involve using these generative AI models to produce attack trees with a root attack and a set of connected sub-attacks from system specifications. However, our initial attempts at exploring this method often produced disappointing results. The LLMs we performed initial tests with, using just custom prompts, frequently suggested elementary attacks that did not align with strategic criminal objectives. In addition, these LLMs oftentimes suggested sub-attacks that were irrelevant to the provided system specification. Our approach, therefore, combines the use of LLMs, MITRE's Common Attack Pattern Enumeration and Classification (CAPEC) database that contains common attack patterns, and other natural language processing (NLP) models and techniques to identify relevant root attacks and sub-attacks for a provided system specification and diagram them in an attack tree.

The paper begins with a review of related work that examines traditional methodologies for attack tree generation and highlights the limitations inherent in these approaches (section 2). Subsequent sections detail our contributions, focusing on how CAPECs, LLMs, and different kinds of NLP models and techniques such as sentence embedding can be tailored to model complex security scenarios effectively. We present a framework, implemented within TTool, that leverages generative AI with a set of pertinent CAPECs to automate and augment the creation of attack trees from a system specification. CAPECs are deemed pertinent to the provided system specifications using a combination of different NLP models and techniques further described in section 3.4. We then follow this up with an evaluation of the relevance and efficacy of our proposal. We discuss potential enhancements in efficiency, scalability, and thoroughness that this integration offers. Finally, section 5 concludes the paper.

2 Related Works

2.1 Context on Attack Trees

Attack trees [23] are a formalism extensively utilized in cybersecurity risk analysis. They offer a comprehensible structure consisting of nodes that succinctly describe attack steps along with logical operators. This structure allows engineers from diverse backgrounds to readily analyze them and contribute to their extensive application in various contexts ranging from safety-critical systems such as

Supervisory Control and Data Acquisition (SCADA) systems [8,25] and healthcare systems [24,12], to IT systems such as banking systems [10]. Attack trees are utilized across different stages of a system's lifecycle, ranging from system design [22,18] to its operational phase, including facilitating the dynamic reconfiguration of networks [20]. The contribution we present in this paper specifically aims to facilitate attack tree-based security analysis conducted early in the system design cycle, which is crucial for avoiding costly re-engineering during the design and development process [11].

2.2 Attack Tree Generation

Due to the widespread use of attack trees and the potential flaws induced by manual generation by security experts (which is expertise-dependent, tedious, and error-prone) [27], automatic generation of attack trees is not a new research issue. Several literature reviews, including one published this year, are dedicated to this topic [27,13]. Among the attack tree generation approaches reviewed by [27], a common feature is that they all require as input a description of the system provided in a specified formalism (e.g., labeled transition systems or value-passing quality calculus [26]). This is an advantage because the system is described more rigorously, with a high level of detail about its behavior, communications, and architecture. However, this approach also has drawbacks, as it is advantageous to start the risk analysis in parallel with the system design, i.e., before the first models of the system are constructed. In this context, attack trees document plausible attack scenarios based on the provided specifications, aiding the initial system modeling by incorporating these threat scenarios. As the design process progresses, the attack trees are incrementally refined alongside the model, becoming more detailed as the system design takes shape.

Overall, approaches to vulnerability and attack scenario detection often rely on constrained input formalisms. For example, in [15], the authors propose modeling industrial control systems using SysML, which is then converted into the IDP [9] formal framework. This converted model forms the basis for the automatic detection of vulnerabilities and attacks. The detection process is conducted by an analyzer that leverages a dataset of vulnerabilities from ICS-CERT, along with guidelines from NIST and other relevant sources. In [3], the authors propose a methodology that involves consecutively modeling a Cyber-Physical System (CPS) in SysML, converting it into a graph, and then identifying vulnerabilities and attack scenarios using Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE), and CAPEC databases. Closer to the goal of incorporating threat analysis early in the development cycle, Kammüller et al. (2019) propose an approach that incrementally refines specifications through the use of attack trees to identify risks and vulnerabilities. These attack trees are constructed from SysML models that are derived from the initial specifications.

The survey by Konsta et al. [13] reviews 22 approaches for generating attack trees/graphs. Most of these approaches also rely on system descriptions provided in a formalism that is either constrained or requires details about the actual system architecture (such as network topology). However, one paper [14]

does not impose any constraint on the input description of the system. This paper proposes an attack tree template and method to assist security experts in designing their attack trees. We believe that our contribution, which proposes a fully automated method for attack tree generation from a system specification, usefully complements this approach. In addition, as highlighted in [13], a key issue in generating attack trees or graphs is ensuring the production of meaningful results. Some of the approaches surveyed in this study achieve this by design. However, to the best of our knowledge, our approach is the first to incorporate a set of objective metrics to assess the quality of the generated attack trees.

2.3 The CAPEC Effort

Our contribution relies on the analysis of *attack patterns*. Attack patterns are defined by Moore et al. [17] as "deliberate, malicious attack[s] that commonly occurs in specific contexts." As to what these contexts are is open to interpretation and can be in a technical environment such as the data that resides in a network flow of an application framework[1] or a general environment such as sensitive information being kept within physical storage.[2] The article also goes on to state that an attack pattern consists in the following elements:

1. The overall goal of the attack specified by the pattern.
2. A list of preconditions necessary for the attack to be executed.
3. The steps for carrying out the attack.
4. A list of postconditions that are true if the attack is successful.

The CAPEC database [4], currently maintained by The MITRE Corporation, maintained by The MITRE Corporation, provides a continuously updated list of attack patterns based on this definition. These patterns are organized into four abstraction levels: Category, Meta, Standard, and Detailed. For our research, we focused on attack patterns at the Standard level, which is defined as "a specific methodology or technique used in an attack."[3] In addition, we only focus on elements 1 and 3 for determining pertinent CAPECs to a provided system specification.

2.4 TTool-AI

TTool[4] is an open-source model-based software and systems engineering (MBSE) toolkit. It supports the modeling of systems with formally defined SysML profiles and extends SysML to incorporate security aspects among other modeling facets [19]. TTool includes algorithms for simulating and formally verifying these models through direct model-checking [7] or with the use of ProVerif [16], and for generating source code from these models. Additionally, TTool supports the

[1] `https://capec.mitre.org/data/definitions/384.html`
[2] `https://capec.mitre.org/data/definitions/406.html`
[3] `https://capec.mitre.org/about/glossary.html#Standard_Attack_Pattern`
[4] `https://ttool.telecom-paris.fr`

modeling of attack trees [1] to guide MBSE methods that integrate security, such as SysML-Sec [22].

Recently, TTool has been enhanced with an AI-based extension called TTool-AI [2]. TTool-AI is a LLM-based modeling assistant that enables users to automatically generate models from textual specifications, modify them based on natural language queries, or classify requirements from specifications. The accuracy of the LLM's responses is ensured through an automated feedback loop that detects errors and inconsistencies and iteratively asks the LLM to refine the generated models.

Given these features, we have chosen to utilize TTool to implement our contribution as a new feature of TTool-AI.

3 Contributions

3.1 Preliminary Definitions

Definition 1 (Alphabet, Words and Sentences)

- $\mathcal{A} = \{a, A, b, B, \ldots, z, Z\} \cup \{\_\}$ *is the* alphabet.
- $\mathcal{A}*$ *is the set of all finite words generated by* $\mathcal{A}$[5].
- $\epsilon \in \mathcal{A}*$ *is the empty word.*
- $\mathcal{S}$ *is the set of all finite* sentences *on words of* $\mathcal{A}*$*, a sentence being a sequence of words separated by blank spaces or punctuation marks.*

In this paper, we rely on the attack trees introduced in [1]. Below, we provide a formal definition for these trees.

Definition 2 (Attack Trees)
An attack tree *is a 4-uple* $\langle (v_0, V, E), \mathsf{name}, \mathsf{desc}, \mathsf{rank} \rangle$ *where:*

- (v_0, V, E) *is a non-empty, finite, directed rooted tree.*
- $V = V_A \sqcup V_O$[6] *is a set of vertices,* V_A *being a set of* attack vertices *and* V_O *a set of* operator vertices*. Elements of* V_O *are typed by* $\mathsf{type_V} : V_O \rightarrow \{AND, OR, SEQ\}$.
- $v_0 \in V_A$ *is the root of* (v_0, V, E).
- $E \subset V^2 \setminus (V_A^2 \cup V_O^2)$ *is a set of edges, directed towards the root. It is such that* $\forall v_a \in V_A, card(\{v | (v, v_a) \in E\}) \leq 1$
- $\mathsf{name} : V_A \rightarrow \mathcal{A}*$ *is a function that assigns a name to each attack vertex.*
- $\mathsf{desc} : V_A \rightarrow \mathcal{S}$ *is a function that assigns a description to each attack vertex.*
- $\mathsf{rank} : V_A \times (E \cap V_A \times \{v \in V_O | \mathsf{type_V}(v) = SEQ\}) \rightarrow \mathbb{N}$ *is a function that assigns a rank to each child of a* SEQ *node. For* $v_{a1}, v_{a2} \in V_A$ *and* $v_o \in V_O$ *such that* $(v_{a1}, v_o) \in E$ *and* $(v_{a2}, v_o) \in E$*,* $\mathsf{rank}(v_{a1}) < \mathsf{rank}(v_{a2})$ *means that the attack modeled by* v_{a1} *is executed prior to the one modeled by* v_{a2}.
- *a leaf of* (v_0, V, E) *is necessarily an element of* V_A.

[5] $*$ is the Kleene star operator.
[6] "$\sqcup$" denotes the disjoint union operator.

Definition 3 (Attack Patterns)
Let $\mathcal{P} \subset \mathcal{A}* \times \mathcal{S}*$ *be the set of* attack patterns. *An attack pattern is an ordered pair composed of a name and a description. Its name is unique:* $\forall (n, d_1) \in \mathcal{P}, \nexists (n, d_2) \in \mathcal{P}$.

Definition 4 (System Specification)
A system specification *is a text in natural language that provides a description of the functional and architectural aspects of a system.*

3.2 Metrics

Konsta et al. highlights in [13] that a key issue in diagramming attack graphs is ensuring that a graph produces meaningful results. Thus, to capture and assess the quality of produced attack trees in a format that can be quantitatively analyzed, our contribution also includes a set of specially crafted metrics.

TTool, due to the limited number of operators available for constructing attack trees [1], ensures that both human users and LLMs integrated in TTool-AI generate at least a directed graph when creating attack trees. This graph, we call a *generated attack graph* in the rest of the paper, is of the form $\langle (V, E), \mathsf{name}, \mathsf{desc}, \mathsf{rank} \rangle$ be 4-uple where (V, E) is a directed graph, $V = V_A \sqcup V_O$, V_A (resp. V_O) is a set of attack (resp. operator) nodes, elements of V_O are typed by $\mathsf{type_V} : V_O \rightarrow \{AND, OR, SEQ, XOR, BEFORE, AFTER\}$, $E \subset V^2$, and name, desc and rank are the functions defined in Definition 2. In order to ensure that the generated attack graph is an attack tree as defined per Definition 2, we propose a set of rules to check.

Definition 5 *Complexity*
The complexity of an attack tree $\langle (V, E), \mathsf{name}, \mathsf{desc}, \mathsf{rank} \rangle$ *is given by the value* $C = card(V)$.

Definition 6 *Syntax Rules for Attack Vertices*

1. $\mathsf{name} : V_A \rightarrow \mathcal{A}* \setminus \{\epsilon\}$ *(an attack vertex shall have a non-empty name)*
2. $\mathsf{desc} : V_A \rightarrow \mathcal{S} \setminus \{\epsilon\}$ *(an attack vertex shall have a non-empty description)* $\exists v_0 \in V_A$ *such that:*
3. $\forall v_a \in V_A \setminus \{v_0\}, card(\{v | (v_a, v) \in E\}) = 1 \wedge E \cap V_A^2 = \emptyset$ *(an attack vertex shall be connected to a unique parent operator vertex, except if this is the root attack)*
4. $\forall v \in V, \{(v_0, v)\} \cap E = \emptyset$ *(a root attack shall not be connected to a parent operator)*
5. $\forall v \in V \setminus \{v_0\}, (v, v_0) \in E^+$ *where* E^+ *is the transitive closure of* E *and the path between* v *and* v_0 *is unique (each attack vertex shall be connected by exactly one path to the root attack).*

Definition 7 *Syntax Rules for Operator Vertices*

1. $\forall v_o \in V_O, card(\{v|(v, v_o) \in E\}) \geq 2 \wedge E \cap V_O^2 = \emptyset$ *(an operator vertex must have at least two children attack vertices)*
2. $\forall v_o \in V_O, card(\{v|(v_o, v) \in E\}) = 1 \wedge E \cap V_O^2 = \emptyset$ *(an operator vertex must have only one parent attack vertex)*
3. *Elements of* V_O *are typed by* $\mathsf{type_V} : V_O \rightarrow \{AND, OR, SEQ\}$ *(an operator vertex must have only one of the following types: SEQUENTIAL, AND, or OR)*

Based on these rules, we can define a syntactic correctness score:

Definition 8 *Syntactic Correctness Score*
Let $err_a : V_A \rightarrow \mathbb{N}$ *be a function that represents, for each attack vertex, the number of syntax rules that are not respected, and let* $err_o : V_O \rightarrow \mathbb{N}$ *be its corollary for the operator vertices. The syntactic correctness score of a generated attack graph is the value*

$$\frac{\frac{\sum_{v_a \in V_A} err_a(v_a)}{nb.\ of\ att.\ node\ rules} + \frac{\sum_{v_o \in V_O} err_o(v_o)}{nb.\ of\ op.\ node\ rules}}{C}.$$

We propose two additional scores to evaluate the semantic correctness and completeness of an attack tree. Please note that unlike the previous score, these two rely on elements that depend on the evaluator's judgment.

Definition 9 *Semantic Correctness Score*
We denote with r_a *(resp.* r_o*) the number of* relevant *attack (resp. operator) vertices. The semantic correctness score of a generated attack graph is the value*

$$\frac{r_a + r_o}{C}.$$

Definition 10 *Completeness Score*
Let n *represent the number of missing (attack and operator) vertices in an attack tree that are required to cover all possible scenarios leading to its root attack. The completeness score of a generated attack graph is then defined as*

$$\frac{r_a + r_o}{r_a + r_o + n}.$$

3.3 The Attack Tree Generation Process

Our contribution extends the LLM-based block and state-machine diagram generation process introduced in [2] to attack trees. This process leverages two same key mechanisms: prior knowledge embedding and an automated feedback loop. Figure 1 presents an overview of the attack tree generation process, which proceed as follows:

1. The user provides a system specification to the graphical user interface, which can be in any format, including natural language.
2. (Optional) The system specification is processed by a CAPEC tracer, which analyzes it and extracts a list of relevant attack patterns from MITRE's CAPEC database. Our CAPEC tracer relies on comparing sentence embeddings from the system specification with each Standard CAPEC to determine which CAPECs are most relevant to the specification.

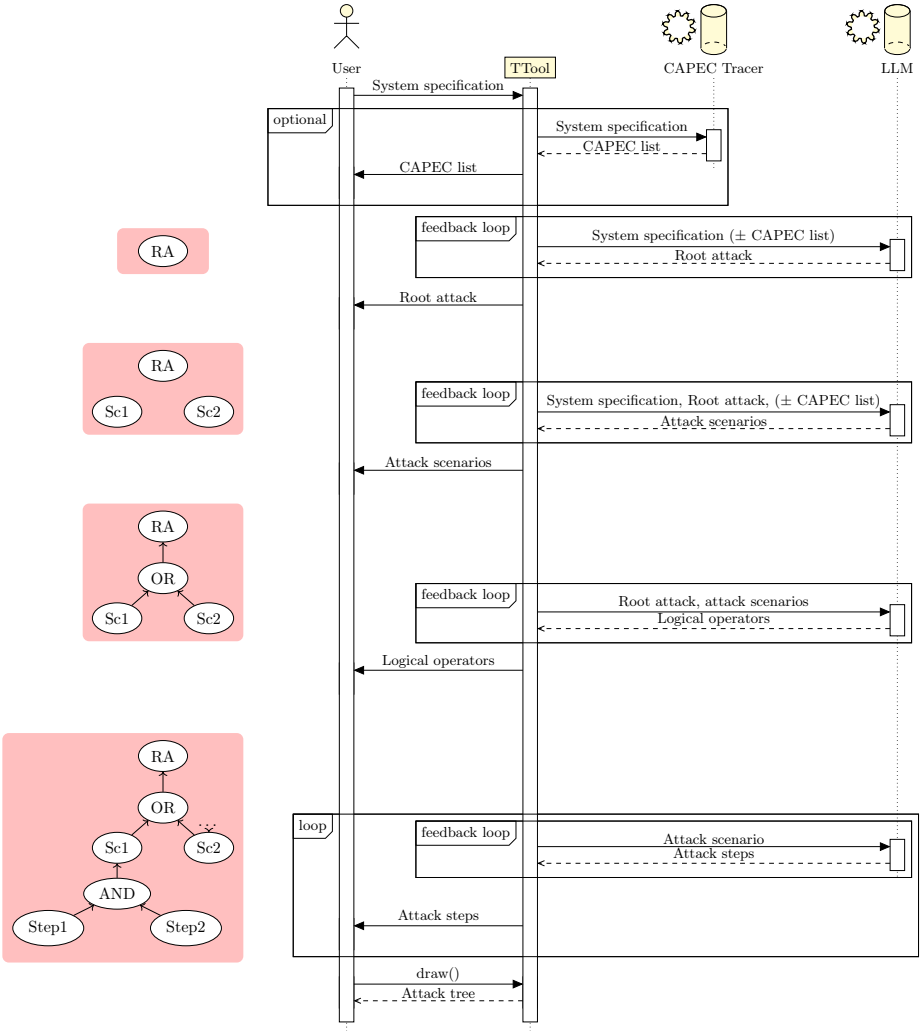

Fig. 1. Sequence diagram of the generation process – CAPEC option enabled

3. Generating v_0. The system specification, along with the optional attack patterns list, is sent to the LLM along with a set of constraints that define the expected format for the root attack, including rules 1 and 2 in definition 6. The LLM identifies a root attack, which is an attack vertex modeling a high-level attack goal targeting the system. The LLM response is then algorithmically checked to determine whether it suits the format constraints: if it is not the case, a new request including the detected errors is sent back to the LLM. This process continues until the LLM produces a correct root attack node.

4. Completing V_A. The tool then provides the LLM with the system specification, root attack, and output format constraints, and tasks it with generating a set of attack scenarios—attack vertices at the immediate lower abstraction level relative to the root attack. A feedback loop ensures the correctness of the attack scenarios' format.
5. Adding the first operators to V_O. The tool tasks the LLM with identifying logical operators that connect the root attack to the attack scenarios. The LLM is provided with the root attack, attack scenarios, and constraints on the expected format, including rules 4 in definition 6 and 1 in definition 7, with here again a feedback loop to ensure format correctness.
6. Completing V_A and V_O. For each attack scenario, the tool sends the scenario and format constraints to the LLM, asking it to generate a set of attack steps, which are attack vertices of lower abstraction. For each attack scenario, the LLM answers also undergo a feedback loop for ensuring attack steps format correctness.
7. The user can choose to ask the LLM to refine the identified steps or to draw the attack tree. If the user opts to draw the attack tree, the tool automatically generates it from the successive responses provided by the LLM.

3.4 Implementation

For this paper we implemented two versions of the attack tree generation framework, one of which integrates selected CAPECs into the process. The version that integrates CAPECs is denoted as *ATGC* and the version that does not is denoted as *ATG*. In addition, we use OpenAI's GPT-4 Turbo model as the LLM to generate each portion of the attack tree.

For *ATGC*, the process can be thought of as two different sub-processes as shown in Figure 2. The first sub-process is the CAPEC Tracer that generates a ranked list of most relevant CAPECs to a provided system specification. To develop the list of pertinent CAPECs for a provided system specification, the first step involves obtaining all Standard CAPECs from the MITRE[7] catalog and decomposing the metadata of each CAPEC. Only Standard CAPECs that are not listed as Obsolete nor Deprecated are parsed. We denote this set of Standard CAPECs as *SC*.

For each CAPEC in *SC*, a list of sentences is generated by combining the description and execution flows of the CAPEC and then sentence tokenizing them using the Natural Language Toolkit (NLTK) Python library [6]. Each sentence is also preprocessed to ensure that only alphanumeric characters and spaces are preserved. The sentences of the system specification are also tokenized using NLTK and preprocessed in the same way as the CAPECs.

Each system specification sentence and CAPEC sentence, respectively, are then converted into a sentence embedding. The list of sentence embeddings associated with a system specification is denoted as *SES* and the list of sentence

[7] `https://capec.mitre.org/data/downloads.html`

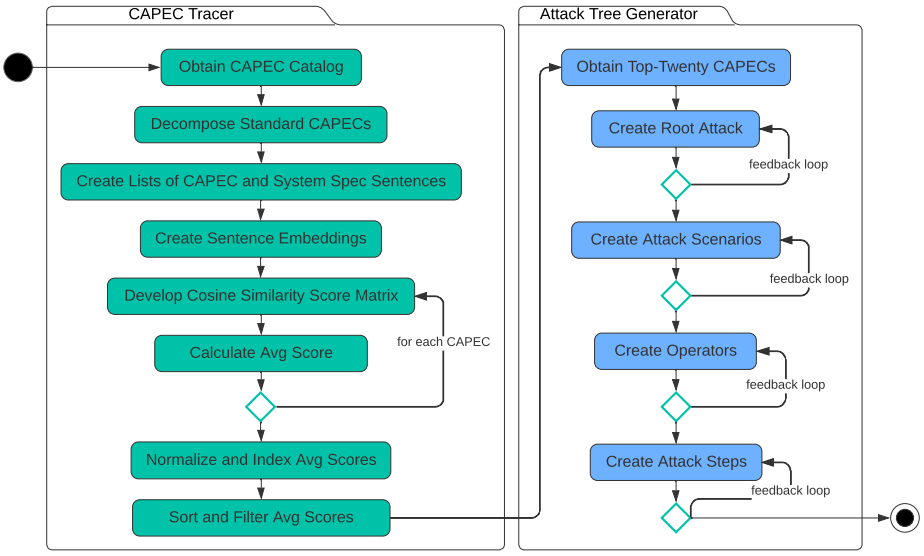

Fig. 2. Activity diagram of the ATGC process

embeddings associated with a CAPEC are denoted as *SEC*. Sentence embeddings are generated using a pretrained Sentence Transformer model [21] known as ATT&CK BERT [5]. To compare how relevant a specific CAPEC is to the system specification, a two-dimensional matrix of cosine similarity scores is constructed, denoted as *CM*, where each element in *CM* is a cosine similarity score between a sentence embedding in *SES* and a sentence embedding in *SEC*. Thus, *CM* represents the cosine similarity scores of all possible pairs of sentence embeddings between *SES* and *SEC*. The arithmetic mean is then calculated among all of the cosine scores in *CM* to produce an averaged cosine similarity score that represents how relevant a CAPEC is to the system specification. This process is repeated for each CAPEC in *SC* that results in a sorted list of indexed scores. The list is filtered out for scores less than or equal to zero. In addition, each score is normalized using min-max normalization, where the min is set to zero and the max is set to the highest score in the list rounded up to the nearest hundredths place, and then multiplied by one hundred. We denote this new converted score in our implementation as a confidence score.

The second sub-process of *ATGC* is the Attack Tree Generator that takes the top twenty CAPECs with the highest confidence scores provided by the CAPEC Tracer and then generates the amount of attack trees needed for the provided system specification as specified by the user. The generation process for each attack tree is essentially performed as described in steps 3 through 7 in section 3.3.

ATG works fundamentally the same as *ATGC*. The main difference with this version is that it solely includes the Attack Tree Generator sub-process and that

the custom LLM prompts used in this sub-process do not include any CAPEC information in them. They are simply designed to ask the LLM for specific parts of the attack tree and apply corrections as needed.

4 Evaluation

This section details the evaluation of our attack tree generation process. It begins with a description of our experimental setup, followed by a summary of the results obtained.

4.1 Experimental Setup

We have evaluated our attack tree generation approach by comparing the attack trees produced by three security engineers of varying experience levels, alongside those generated by our TTool-AI extension (both with and without the CAPEC tracer enabled). The input consisted of three specifications generated by GPT-4 Turbo: one detailing a cloud service infrastructure, another describing a social network mobile application, and the third describing a CPU.

In detail, each engineer was tasked with creating as many attack trees as possible for one of the specifications within a 1.5-hour time limit. For each specification, we also tasked our TTool-AI extension to generate the same number of attack trees, with and without the CAPEC tracer enabled. For the cloud service, five attack trees were generated by the engineer and ten by TTool-AI. For the social network application, one tree was created by the engineer and two by TTool-AI. Similarly, for the CPU system, one tree was generated by the engineer and two by TTool-AI.

The attack trees were then graded using the metrics defined in Section 3.2. For the assessment of the handmade attack trees, each engineer graded the trees created by one of their colleagues. The grading process was organized in such a way that no engineer graded the work of the engineer who evaluated their attack tree. Furthermore, to limit the number of missing nodes identified during the evaluation of a tree's completeness (see Definition 10), the grading time for each case study was restricted to 30 minutes.

Reproducibility : all the attack trees, detailed gradings, and input specifications are publicly available in an anonymous GitHub repository.[8]

4.2 Results

Table 1 provides a synthesis of the attack trees grading and Figures 3 and 4 show example branches of a manual and an *ATGC* attack tree respectively.[9] [10]

[8] `https://github.com/zebradile/ttool-ai/tree/main/attacktrees`

[9] For Figures 3 and 4, due to space constraints these screenshots only show a sample of attack scenarios and their children attack steps while in the actual diagrams there are additional attack scenario branches.

[10] The children of a SEQUENCE operator node are meant to be read from left to right in terms of execution order.

Table 1. Evaluation results

(a) Cloud service case study

Creator	Complexity	Semantic correctness	Completeness	Syntactic correctness	Generation time (s)
Engineer	26	1	1	0.99	5400
	19	1	0.83	1	
	19	1	1	1	
	15	1	1	1	
	8	1	0.8	0.93	
ATG	31	0.74	0.88	1	400
	28	0.89	0.86	1	
	19	0.63	0.86	1	
	23	0.48	0.79	1	
	23	0	0	1	
ATGC	12	0.92	0.85	1	549
	15	1	1	1	
	19	0.53	0.77	1	
	14	0.93	0.93	1	
	28	0.93	0.81	1	

(b) CPU case study

Creator	Complexity	Semantic correctness	Completeness	Syntactic correctness	Generation time (s)
Engineer	15	1	0.83	0.99	5400
ATG	26	0.5	0.87	1	103
ATGC	15	1	0.79	1	140

(c) Social network application case study

Creator	Complexity	Semantic correctness	Completeness	Syntactic correctness	Generation time (s)
Engineer	20	1	0.77	0.88	5400
ATG	30	0.77	0.88	1	91
ATGC	18	1	0.9	1	127

(d) Statistics

Creator		Complexity	Semantic correctness	Completeness	Syntactic correctness	Generation time (s)
Engineer	Average	17.4	1	0.89	0.97	2314.29
	Std. Dev.	5.2	0	0.1	0.04	—
ATG	Average	25.7	0.57	0.73	1	84.86
	Std. Dev.	4.31	0.27	0.33	0	—
ATGC	Average	17.29	0.9	0.86	1	116.57
	Std. Dev.	4.9	0.16	0.08	0	11.26

We also want to highlight that the descriptions generated for the attack nodes from each attack tree could not be shown in these figures also due to space constraints. However as mentioned in section 4.1, all of our results, including these descriptions, can be viewed in the anonymous GitHub repository.

From these results, we have deduced the following observations:

- *ATGC* and *ATG* will always be syntactically correct and will always guarantee to follow the logical structure of a defined attack tree schema, due to the feedback loops put in place. On the other side, we see that oftentimes manual attack trees have either syntactic and/or structural errors in them.
- *ATGC* and *ATG* took much less time to generate attack trees compared to the manual attack trees.
- Manual attack trees on average are more robust compared to the attack trees generated by *ATGC* and *ATG*, particularly with generating more relevant attack steps that describe the full process of their parent attack scenarios.

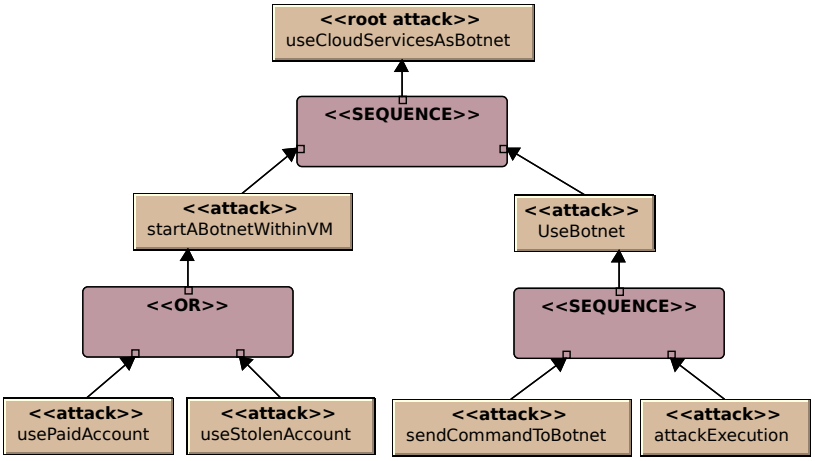

Fig. 3. Screenshot showing two example branches of a manual attack tree produced for the cloud service

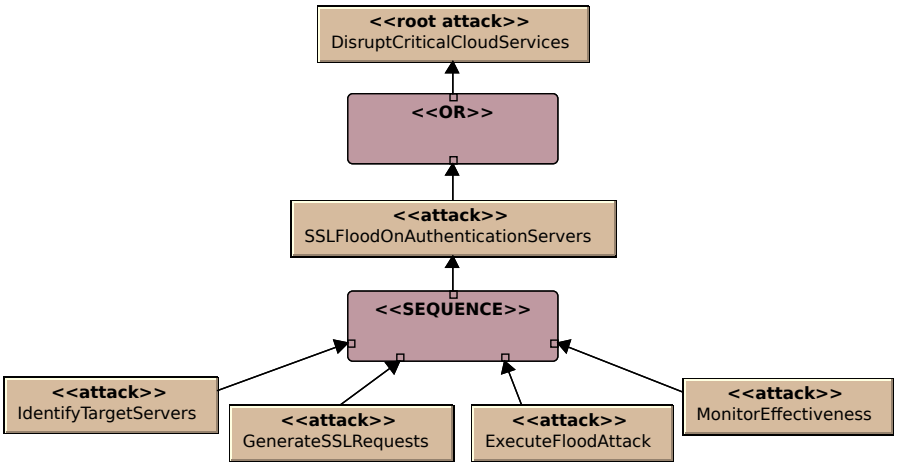

Fig. 4. Screenshot showing one example branch of an ATGC attack tree produced for the cloud service

- *ATG* generates too general attack trees that oftentimes have attack/operator nodes that are not relevant to the root attack as seen with their semantic correctness and completeness scores.
- *ATGC* generates attack trees with relevant attack scenarios most of the time, which are often on par with manual attack trees in terms of semantic correctness. However, it needs improvement in terms of creating sets of attack steps that fully describe what an attacker needs to achieve the attack scenario.

- *ATG* on average generates attack trees that have higher complexity scores compared to *ATGC*, but this also coincides with the former having much less semantic correctness scores than the latter.
- In one of the attack trees that *ATG* generated for the cloud system, none of the attack nodes, including the root node, was relevant to the system. Thus, this tree was given a zero for semantic correctness and completeness.
- *ATG* is on average faster at creating attack trees compared to *ATGC*. When creating five attack trees for the cloud system however, *ATG* took longer to finish all five trees compared to *ATGC* since it produced errors that the feedback loops needed to correct. *ATGC* when creating each of the five attack trees for the cloud system produced each part of the tree correctly the first time.
- Analysis of the metrics (see Table 1 (d)) allows for several conclusions. We see that the performance of ATG is good, but the standard deviations indicate a fairly variable quality from one attack tree to another. In addition, ATGC achieves excellent scores with low standard deviations, indicating consistent high quality across all generated trees. This reliability allows engineers to use these trees with a high degree of confidence. Generally speaking, we can observe that the quality of trees generated by engineers is close to that of trees from ATGC. In addition, ATGC has an average time that is 20 times lower than that of an engineer. Finally, we can note that syntax correction is ensured in all cases by ATG(C).

5 Discussion and Conclusions

For our research, we explored the capability of LLMs to produce complete, semantically correct, and sytaxically correct attack trees given a textual system specification. We have observed this potential increase when supplied with additional cyber security data such as CAPECs. Although our framework does not produce attack trees that are as robust as those created by security experts, it delivers fair results in a much shorter time compared to attack trees initially diagrammed by an expert. Thus, we have confidence that this new feature of TTool-AI produces attack trees that are solid foundations for experts to further refine and enhance.

We also consider that there are lots of areas to further explore with this research to improve on the completeness, semantic, and sytaxic correctness of LLM generated attack trees. One such area is producing attack trees with different LLMs as we have only tested attack trees with GPT-4 for our research. In addition, there could be further improvements made for the CAPEC tracer and Attack Tree Generator sub-processes. For the CAPEC tracer, we would need to explore additional NLP models and techniques to improve the CAPEC ranking process. Also, there could be other cyber security catalogs such as MITRE's ATT&CK matrix whose data would either help improve the Attack Tree Generator in addition to CAPECs or even perhaps instead of CAPECs. For the Attack Tree Generator sub-processes, additional work could be done to improve

the clarity of the prompts and the way that attack trees are constructed such that it would be more straightforward for an LLM to produce more robust attack trees that better model the motivation of an attack and the steps to achieve said attack given a system specification.

References

1. Apvrille, L., Roudier, Y.: SysML-Sec attack graphs: compact representations for complex attacks. In: Graphical Models for Security: Second International Workshop, GraMSec 2015, Verona, Italy, July 13, 2015, Revised Selected Papers 2. pp. 35–49. Springer (2016)
2. Apvrille, L., Sultan, B.: System Architects Are not Alone Anymore: Automatic System Modeling with AI. In: 12th International Conference on Model-Based Software and Systems Engineering. pp. 27–38. SCITEPRESS-Science and Technology Publications (2024), Best paper award
3. Bakirtzis, G., Carter, B.T., Elks, C.R., Fleming, C.H.: A model-based approach to security analysis for cyber-physical systems. In: 2018 Annual IEEE International Systems Conference (SysCon). pp. 1–8. IEEE (2018)
4. Barnum, S.: Common Attack Pattern Enumeration and Classification (CAPEC) Schema Description. Cigital (2008)
5. Basel, A., Al-Sheer, E., Singhal, A., Khan, L., Hamlen, K.: Smet: Semantic mapping of cve to att&ck and its application to cyber security. No. 13942, DBSec 2023: Data and Applications Security and Privacy XXXVII, Sophia Antopolis, FR (2023-07-12 04:07:00 2023). `https://doi.org/https://doi.org/10.1007/978-3-031-37586-6_15`, `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=936761`
6. Bird, S., Klein, E., Loper, E.: Natural language processing with Python: analyzing text with the natural language toolkit. " O'Reilly Media, Inc." (2009)
7. Calvino, A.T., Apvrille, L.: Direct model-checking of SysML models. In: 9th International Conference on Model-Driven Engineering and Software Development. pp. 216–223. SCITEPRESS-Science and Technology Publications (2021)
8. Cherdantseva, Y., Burnap, P., Blyth, A., Eden, P., Jones, K., Soulsby, H., Stoddart, K.: A review of cyber security risk assessment methods for SCADA systems. Computers & Security **56**, 1–27 (2016). `https://doi.org/https://doi.org/10.1016/j.cose.2015.09.009`, `https://www.sciencedirect.com/science/article/pii/S0167404815001388`
9. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system. In: Declarative Logic Programming: Theory, Systems, and Applications, pp. 279–323 (2018)
10. Edge, K., Raines, R., Grimaila, M., Baldwin, R., Bennington, R., Reuter, C.: The use of attack and protection trees to analyze security for an online banking system. In: 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07). pp. 144b–144b. IEEE (2007)
11. Glumich, S., Riley, J., Ratazzi, P., Ozanam, A.: BP: Integrating Cyber Vulnerability Assessments Earlier into the Systems Development Lifecycle. In: 2018 IEEE Secure Development Conference
12. Kammüller, F.: Combining secure system design with risk assessment for IOT healthcare systems. In: 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). pp. 961–966. IEEE (2019)

13. Konsta, A.M., Lafuente, A.L., Spiga, B., Dragoni, N.: Survey: Automatic generation of attack trees and attack graphs. Computers & Security **137**, 103602 (2024)
14. Kumar, R.: An attack tree template based on feature diagram hierarchy. In: 2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys). pp. 92–97. IEEE (2020)
15. Lemaire, L., Lapon, J., Decker, B.D., Naessens, V.: A SysML extension for security analysis of industrial control systems. In: 2nd International Symposium for ICS & SCADA Cyber Security Research 2014 (ICS-CSR 2014). BCS Learning & Development (2014)
16. Lugou, F., Li, L.W., Apvrille, L., Ameur-Boulifa, R.: SysML models and model transformation for security. In: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 331–338. IEEE (2016)
17. Moore, A., Ellison, R., Linger, R.: Attack modeling for information security and survivability. Tech. Rep. CMU/SEI-2001-TN-001 (Mar 2001), `https://doi.org/10.1184/R1/6572063.v1`
18. Paul, S.: Towards automating the construction & maintenance of attack trees: a feasibility study. arXiv preprint arXiv:1404.1986 (2014)
19. Pedroza, G., Apvrille, L., Knorreck, D.: AVATAR: A SysML environment for the formal verification of safety and security properties. In: 2011 11th Annual International Conference on New Technologies of Distributed Systems. pp. 1–10. IEEE (2011)
20. Rehák, M., Staab, E., Fusenig, V., Pěchouček, M., Grill, M., Stiborek, J., Bartoš, K., Engel, T.: Runtime monitoring and dynamic reconfiguration for intrusion detection systems. In: Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings 12. pp. 61–80. Springer (2009)
21. Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics (11 2019), `http://arxiv.org/abs/1908.10084`
22. Roudier, Y., Apvrille, L.: SysML-Sec: A model driven approach for designing safe and secure systems. In: 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 655–664. INSTICC (2015)
23. Schneier, B.: Attack trees. Dr. Dobb's journal **24**(12), 21–29 (1999)
24. Siddiqi, M.A., Seepers, R.M., Hamad, M., Prevelakis, V., Strydis, C.: Attack-tree-based threat modeling of medical implants. In: PROOFS. pp. 32–49 (2018)
25. Ten, C.W., Liu, C.C., Govindarasu, M.: Vulnerability assessment of cybersecurity for SCADA systems using attack trees. In: 2007 IEEE Power Engineering Society General Meeting. pp. 1–8. IEEE (2007)
26. Vigo, R., Nielson, F., Nielson, H.R.: Uniform protection for multi-exposed targets. In: Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings 34. pp. 182–198. Springer (2014)
27. Wideł, W., Audinot, M., Fila, B., Pinchinat, S.: Beyond 2014: Formal Methods for Attack Tree–based Security Modeling. ACM Computing Surveys (CSUR) **52**(4), 1–36 (2019)