From Attack Trees to Attack-Defense Trees with Generative AI & Natural Language Processing

Alan Birchler De Allende
alan.birchler@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Sophia-Antipolis, France

Bastien Sultan
bastien.sultan@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Sophia-Antipolis, France

Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Sophia-Antipolis, France

ABSTRACT

Attack-defense trees, an extension of attack trees, are extensively used by security engineers to document potential countermeasures for security threats present in a system's design. These trees help integrate initial system models with countermeasures, allowing for early testing of their efficiency and impact in the design cycle. Despite advancements in automating attack tree construction, selecting the initial set of countermeasures for conversion into an attack-defense tree remains largely manual. This paper proposes an approach and a tool that extends the TTool-AI attack tree generation feature by leveraging large language models and natural language processing to create a set of countermeasures and generate attack-defense trees based on an input attack tree. To evaluate our contribution, our approach is tested using attack-defense trees generated from attack trees, each representing possible threats to an associated system specification. In addition, we introduce metrics to assess the semantic correctness and completeness of the generated attack-defense trees. We compared, using our metrics, the attack-defense trees created from our methodology to those created by an engineer and found that attack-defense trees created using AI and secondary mitigation data provided better trees than solely using AI. We also discovered that this approach generated trees that were comparable to the quality of attack-defense trees generated from a security engineer at the associate level. From these results, we believe that our contribution could aid engineers in identifying not only appropriate countermeasures for attack trees but also the optimal number of countermeasures, avoiding the complexity of redundant mitigations. Furthermore, our approach complements standard modeling practices, particularly during the initial design phase, reducing the need for time-consuming re-engineering throughout the system's lifecycle.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computing methodologies → Model development and analysis**; **Artificial intelligence**.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687804

KEYWORDS

Artificial Intelligence, Large-Language Models, Attack-Defense Trees, Model-Driven Engineering

ACM Reference Format:
Alan Birchler De Allende, Bastien Sultan, and Ludovic Apvrille. 2024. From Attack Trees to Attack-Defense Trees with Generative AI & Natural Language Processing. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652620.3687804

1 INTRODUCTION

In the evolving landscape of cybersecurity, whether it concerns information technology (IT) systems or embedded systems, the dynamic and sophisticated nature of threats requires an accurate selection of the necessary countermeasures. Traditionally, security analysts have employed attack trees to visualize potential attack scenarios: these scenarios are expected to support the identification of the corresponding vulnerabilities. However, this identification approach, while comprehensive, often remains labor-intensive and prone to human error, particularly when dealing with complex systems and attack vectors. The advent of generative artificial intelligence (AI) and natural language processing (NLP) presents an opportunity to enhance this traditional manual approach mostly based on expertise. This paper explores the integration of large language models (LLM), which is an application of generative AI technologies, and NLP techniques into the construction of attack-defense trees (ADTs) from pre-existing attack tree diagrams. This contribution complements the attack tree generation method from textual specifications that we previously introduced [1]. It applies similar tree generation and evaluation methodologies, with the following differences:

(1) The graphical output is now an ADT instead of a mere attack tree.
(2) The generation process thus identifies countermeasure leaves and pairs them with attack leaves, rather than producing a complete attack tree structure.
(3) The input includes an attack tree, as opposed to a textual system specification.
(4) The evaluation metrics (see Section 4) have been adapted specifically to ADTs.

Specifically, our contributions focus on the automated identification and inclusion of security mitigations into attack tree diagrams. To achieve this, we explore various automation strategies that leverage both LLMs and existing security countermeasures

databases. Generative AI has proven effective in supporting Model-Based Systems Engineering (MBSE) [3]. However, traditional security architectures often incorporate a semi-automated approach by utilizing regularly updated databases containing information on attacks and their corresponding countermeasures, such as Common Vulnerabilities and Exposures (CVEs) and Common Attack Pattern Enumeration and Classification (CAPECs). These databases provide a continually refreshed resource, contrasting with AI models that require extensive and computationally intensive retraining cycles. Thus, this paper also shows another example of how LLMs and NLP techniques can be combined to better support cybersecurity aspects in MBSE approaches by providing LLMs with additional data pruned with NLP.

The paper is organized as follows. Section 2 introduces the notion of ADTs and CAPECs and provides a short description of our toolkit which we have implemented our work in. Section 3 relates our contribution to similar work and Section 4 presents the heart of our contribution based on CAPECs, AI/LLMs, and NLP techniques. Section 5 evaluates our proposition with regards to, e.g., a manual creation of ADTs. Finally, Section 6 concludes the paper.

2 CONTEXT

2.1 Attack-Defense Trees

Attack trees are conceptual diagrams used often in cybersecurity risk analysis of systems and "provide a formal, methodical way of describing the security of systems, based on varying attacks" [22]. They are formatted as a tree structure such that the attack steps that an attacker needs to complete to achieve a possible attack scenario against a system are outlined as a set of parent/child nodes (also known as attack nodes) connected to one another via edges. This formatting offers a straightforward and comprehensible structure and succinctly describes the steps needed for an attacker to achieve a malicious goal against a system. It also ensures that engineers from diverse backgrounds can readily understand and utilize them for various types of systems such as maritime systems [12], healthcare systems [14, 23], railway systems [9, 16], banking systems [11], etc.

Attack-defense trees are an extension of the attack-tree schema such that they diagram possible defensive action nodes, also known as countermeasures, and connect these nodes to relevant attack nodes on the tree [15]. The connections represent the defenses that can be employed to block an attacker from completing the associated attack steps, thus preventing the overarching attack scenario from being utilized against the system.

2.2 TTool & TTool-AI

TTool[1] is an open-source model-driven engineering toolkit designed for modeling, simulation, formal verification, and code generation. Two such models that users can manually diagram in the toolkit are attack and ADTs. Furthermore, these diagrams are integrated in TTool as part of the model-driven design processes for security-critical systems. TTool supports two model-driven engineering (MDE) methods aimed at secure system design: SysML-Sec [19] and W-Sec [25]. Specifically, W-Sec focuses on selecting the best countermeasures to address a set of attack scenarios, by enabling a comprehensive evaluation of their impact on safety, security, and performance thanks to simulation and formal verification of SysML models.

These methodologies, which rely on incremental modeling and verification of multi-view and often complex system models, have certain limitations. They can be time-consuming for engineers, prone to errors in incremental modifications, and require substantial time for successive model-checking stages. Over the past two years, we have proposed several contributions to address these challenges. These include a mutation language for scripting incremental modifications of SysML models [26], reducing time and error rates for engineers, and a new incremental model-checking algorithm that reduces the complexity of successive proofs by building on previous ones [10]. The contribution presented in this paper continues this effort by addressing the selection of input countermeasures required for executing W-Sec. This is done through the use of an in-house, LLM-based assistant named TTool-AI [2, 3] that already offers multiple functions to aid engineers for the following MDE processes:

- Generating SysML use-case, block, and state-machine diagrams from textual specifications.
- Modifying SysML models from a query written in natural language.
- Detecting and correcting internal and cross-view inconsistencies between different SysML views modeling the same system, a topic we discuss further at this MODELS'24 conference [2].
- Generating attack trees from textual specifications [1].

Thus, the contribution presented in this paper extends the functions of TTool-AI such that this assistant is now also able to aid engineers with the creation of ADTs in the following two ways:

- Identification of relevant countermeasures for given attack trees during the threat analysis of a system.
- Converting said attack trees into ADTs by modeling the identified countermeasures into the tree structure.

2.3 Attack Patterns & CAPECs

Moore et al. [17] delineates an attack pattern as a "deliberate, malicious attack that commonly occurs in specific contexts." Such an attack pattern contains the following elements:

(1) The overall goal of the attack specified by the pattern.
(2) A list of preconditions for its use.
(3) The steps for carrying out the attack.
(4) A list of postconditions that are true if the attack is successful.

The CAPEC database is an inventory of common attack patterns that was originally conceived for the United States Department of Homeland Security in 2007 [5]. It is currently maintained by the MITRE Corporation and has been used for several areas of cyber security research and development such the methodology proposed by Bakirtzis et al. [4] and that involves consecutively modeling a Cyber-Physical System (CPS) in SysML, converting it into a graph, and then identifying vulnerabilities and attack scenarios using the Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE), and CAPEC databases. Attack patterns are

[1] https://ttool.telecom-paris.fr

abstracted in the CAPEC database into four levels: Category, Meta, Standard, and Detailed.[2]

Several attack patterns in the database also include mitigation information for how to prevent an attacker from using them. To convert a provided attack tree into an ADT in TTool, we use CAPEC mitigations relevant to the attack nodes in the attack tree as a secondary source of data to inject into an LLM for the creation of countermeasures. This process is further explained in Sections 4.1 and 4.3. Only attack patterns at the Standard level were used for our contribution, which is defined as "a specific methodology or technique used in an attack."[3]

3 RELATED WORKS

A significant body of research addresses the challenge of selecting an optimal set of countermeasures in ADTs. This optimization problem primarily arises from the need to reduce the cost of mitigations [20]. However, it is equally crucial for preventing unnecessary complexity in the system and avoiding the introduction of a wide variety of new components that could themselves increase the system's attack surface [7] or introduce dependability and performance issues [25]. Overall, existing approaches rely on either a pre-constructed ADT to optimize, or an attack tree along with a set of countermeasures or a pool of known defenses. Thus, the aim with an ADT is to identify a Goldilocks subset of countermeasures that efficiently addresses the attack scenarios depicted by its counterpart attack tree.

For instance, in their paper introducing a threat analysis method based on ADTs, Wang et al. [27] propose an algorithm that constructs a minimal set of countermeasures to cover all attack scenarios depicted in an attack tree. This algorithm takes an ADT as input, which includes a set of possible safeguards.

Addressing this countermeasures optimization problem, Fila et Widel [13] propose a comprehensive (and fully tool-supported) method for optimal countermeasure selection from an ADT, building upon game theory and relying on integer linear programming to solve the optimization problems.

In another effort focused on countermeasure set optimization, Roy et al. [21] propose a process for selecting an optimal set of countermeasures based on attack-countermeasure trees. This process aims to minimize both the number and cost of countermeasures already present in the tree, while ensuring that all attack scenarios are still efficiently covered.

In the context of TTool, Berro et al. [7] propose an optimization approach to identify the optimal subset of countermeasures already present in an ADT. Their approach considers parameters such as the increase in minimal resources required to support the countermeasures and the increase in level of expertise needed from engineers responsible for deploying these security measures.

Stan et al. [24] propose an interesting approach to identify an optimal set of countermeasures from an attack tree. Their method focuses on selecting a subset of countermeasures derived from a predefined set of defense mechanisms that covers most possibilities such as firewalls, software patches, anti-malware software, etc. Our contribution complements Stan et al.'s method by creating countermeasures on potentially more fine-grained models that can be evaluated later on in the development cycle using W-Sec without limiting potential defense mechanisms.

Overall, our approach operates upstream of existing contributions by automatically generating the initial set of countermeasures on the basis of an input attack tree and the system specification. This set can then be optimized using the methods surveyed in this section, as well as MDE approaches like W-Sec [25] or SysML-Sec [19]. This is a topic that, to the best of our knowledge, has not been previously addressed.

4 CONTRIBUTIONS

4.1 Definitions

In this paper, ADTs adheres to the schema defined by Berro et al. [7]. Below, we provide formal definitions that adhere to these ADTs. Note that since the ADT schema is an extension of the attack tree schema, they derive from the attack-tree definitions we have previously introduced [1].

Definition 1 (Alphabet, Words and Sentences).

- $\mathcal{A} = \{a, A, b, B, \ldots, z, Z\}$ *is the* alphabet.
- $\mathcal{A}*$ *is the set of all finite words generated by* $\mathcal{A}$[4].
- $\epsilon \in \mathcal{A}*$ *is the empty word.*
- $\mathcal{S} \subset (\mathcal{A} * \cup \{\_\} \cup \{.,,\})*$ *is the set of all finite* sentences.

Definition 2 (Attack-Defense Trees).
An Attack-defense tree *is a 3-tuple* $\langle (v_0, V, E), desc, rank \rangle$ *where:*

- (V, ra, E) *is a non-empty, finite, directed rooted tree.*
- $V = V_A \sqcup V_O \sqcup V_C$[5] *is a set of vertices,* V_A *being a set of* attack vertices, V_O *a set of* operator vertices *and* V_C *a set of* countermeasure vertices.
- $ra \in V_A$ *is the root of* (V, ra, E). *In the rest of the paper,* ra *is called the* root attack.
- $E \subset V_A \times V_O \sqcup V_O \times V_A \sqcup V_C \times V_A$ *is a set of edges. It is such that* $\forall v_a \in V_A, card(\{v_o | (v_o, v_a) \in E \cap V_O \times V_A\}) \leq 1$ *and* $\forall v_o \in V_O, card(\{v | (v, v_o)\}) \geq 0$.
- $desc : \begin{cases} V_O \rightarrow \{and, or, sequence\} \\ V_A \sqcup V_C \rightarrow \mathcal{A} * \setminus \{\epsilon\} \times \mathcal{S} \setminus \{\epsilon\} \end{cases}$ *is a function that provides a description for each vertex. It associates attack and countermeasure vertices with a name and a textual description, and operator vertices with a type.*
- $rank : V_A \times (E \cap V_A \times \{v \in V_O | desc(v) = sequence\}) \rightarrow \mathbb{N}$ *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* $(v_{a2}, v_o) \in E$, $rank(v_{a1}) < rank(v_{a2})$ *means that the attack modeled by* v_{a1} *is executed prior to the one modeled by* v_{a2}.

We define below a set of metrics to assess the quality of such ADTs. These metrics will provide a basis for the evaluation of our contributions in Section 5.

Definition 3 (ADTs Metrics).
Let $\langle (v_0, V, E), desc, rank \rangle$ *be an ADT. We define its:*

- *Complexity as* $card(E \cap V_C \times V_A)$.

[2] https://capec.mitre.org/about/glossary.html
[3] https://capec.mitre.org/about/glossary.html#Standard_Attack_Pattern
[4] $*$ denotes the Kleene star operator.
[5] "$\sqcup$" denotes the disjoint union operator.

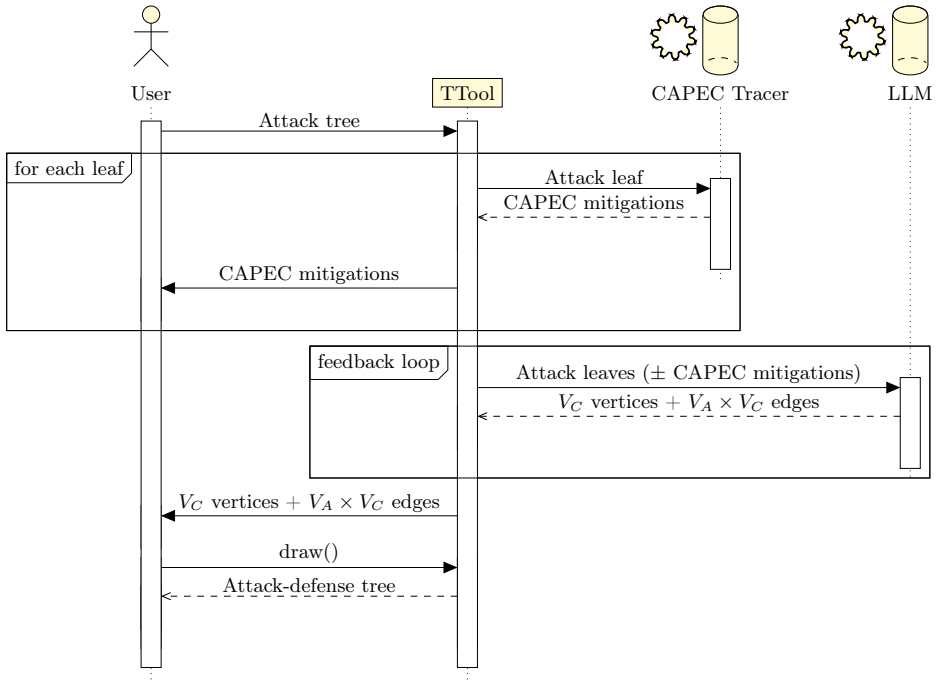

Figure 1: Sequence diagram illustrating our contribution

- *Number of relevant mitigations RM as the number of edges in* $E \cap V_C \times V_A$ *that are relevant.*
- *Number of missing mitigations MM as the number of edges missing in* $E \cap V_C \times V_A$ *to associate an effective set of countermeasure vertices to the attack vertices.*
- *Semantic correctness as* $\frac{RM}{card(E \cap V_C \times V_A)}$.
- *Completeness as* $\frac{RM}{MM+RM}$.

Below, we provide an example of how these metrics are calculated.

Example 1 (ADTs Metrics Example).
We consider the ADT displayed in Figure 3:

- *The ADT includes four edges connecting countermeasures to attack vertices, resulting in a complexity of* 4.
- *The proposed countermeasures appear relevant, so we assign a relevant mitigations score* $RM = 4$.
- *A potential mitigation "Use obfuscated code" could be added to address the first attack step, indicating that there is 1 missing mitigation.*
- *Consequently, the semantic correctness is calculated* $\frac{4}{4} = 1$.
- *And the completeness as* $\frac{4}{4+1} = \frac{4}{5}$.

As pointed out in Section 3, a meaningful ADT is one that has a Goldilocks set of countermeasure to attack node pairings (edges). This is why we established *RM* and *MM* as part of our metrics to denote how far off generated ADTs are from their golden set. Furthermore, semantic correctness and completeness were also established essentially as transformed ratios of *RM* and *MM* to ensure that the quality of generated ADTs could be compared with one another in an unbiased manner since *RM* and *MM* are influenced by the size of the attack tree that an ADT is created from.

4.2 Framework Overview

As mentioned in Section 2.2, our contribution extends TTool-AI using a generation methodology similar to the one we introduced for the creation of ADTs [1]. The input attack trees adhere to the standard structure that Kordy et al. defines [15] where the root of each attack tree corresponds to an attacker's goal, the children of a node in the tree are abstractions of the root node into attack scenarios, and the leaves of the tree are the actions to be executed by an attacker. Since leaf nodes represent the technical steps that an attacker would need to perform to achieve the root goal under this composition, our methodology focuses on using the data of an attack tree's leaf nodes and provides countermeasures only for said leaf nodes. Note that, as stated in Definition 2, all attack nodes in the attack trees used in this paper are associated with a description including a title and a textual description explaining what the node represents.

To see if injecting additional CAPEC mitigation data into the ADT generation methodology would provide better scores for the metrics defined in Section 4.1 and thus overall better ADTs, we conducted implementation tests diverging our methodology into two separate versions, as done for our attack tree generation methodology [1]. In the rest of the paper, the version that includes CAPEC mitigations—ultimately the one we retained—is labeled as **ADTGC**, while the version without these mitigations is labeled as **ADTG**. We used OpenAI's GPT-4 Turbo model as the underlying LLM for countermeasure generation.

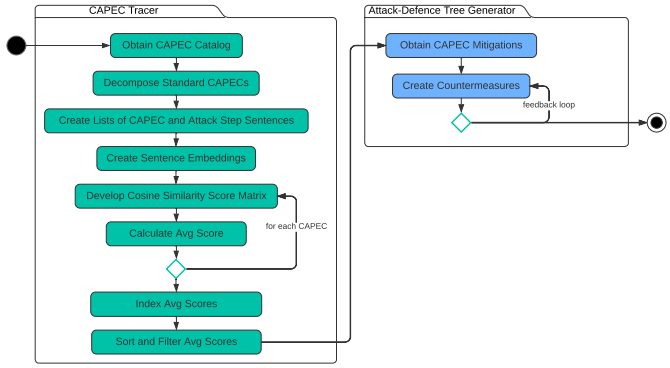

Figure 2: Activity diagram of the ADTGC process. The CAPEC tracer behavior comes from [1].

Figure 1 as well as the following enumerated stages present an overview of the ADT generation methodology:

(1) The user selects, in the TTool GUI, an attack tree diagram that has been modeled with TTool. This attack tree provides either from a system architect, or from an automated generation based on CAPEC or LLMs.
(2) (*ADTGC*) The description of each leaf node in the attack tree diagram is extracted and processed by an in-house CAPEC tracer program. For each provided description, the tracer is responsible for analyzing and relating it to a list of relevant mitigations from MITRE's CAPEC database. The lists of relevant mitigations generated from each leaf node description are then aggregated and de-duplicated, which results in a unique, final list of suggested CAPEC mitigations. This list is displayed in the TTool-AI GUI for user reference. No action is required from the user, as the tool will automatically integrate the list into the request in the next step.
(3) A request is then built and sent to the LLM that includes all leaf node titles and descriptions from the attack tree diagram, where a title and its associated description represent one leaf node. The request also includes a set of constraints, the list of CAPEC mitigations and a question tasking the LLM with the generation of a list of countermeasures and countermeasures-attack edges. The question additionally includes asking the LLM to provide titles and descriptions for each countermeasure. The set of constraints is responsible for ensuring that the LLM formats its response to a standardized and parsable JSON schema.
(4) From there, the LLM identifies a list of countermeasures and associates each of them with at least one leaf node. An automated feedback loop ensures that the proposed countermeasures adhere to the format constraints, iteratively refining the response until it converges to a list of countermeasures that satisfy the constraints or till a maximum of ten retries have occurred.
(5) TTool-AI presents the final list of countermeasure and attack vertex pairings identified by the LLM. If the user is satisfied with the proposed pairings, they can validate the response by clicking a button in the GUI. Upon validation, TTool generates a new ADT. This new diagram is essentially a copy of the original attack tree diagram but includes additional countermeasure nodes connected to the leaf attack nodes.

Note that the folder of the CAPEC tracer program is located in the root directory of the public TTool GitLab repository[6] under the name 'capectracer'.

4.3 Further Implementation Details

The *ADTGC* version, we retained for the final implementation in TTool, consists in two successive sub-mechanisms as shown in Figure 2.

4.3.1 CAPEC Tracer. The first sub-mechanism is the in-house CAPEC Tracer program, mentioned in Section 4.1. We introduced this CAPEC Tracer program for attack tree generation [1], and adapted it to mitigation identification for the needs of the present paper. However, for the sake of self-containment, we provide below the full description of its functionality we previously introduced [1]. The CAPEC Tracer generates a list of relevant CAPEC mitigations for a given leaf node, as detailed in stage 2. This process begins by retrieving all Standard CAPECs from the MITRE database[7] and fragmenting the metadata of each CAPEC. Then, a set of Standard CAPECs, referred to as *SC*, is built by selecting Standard CAPECs that are not marked as *obsolete* or *deprecated*.

After fragmenting the CAPEC data, a list of sentences is constructed by combining the description (element 1 in Moore et al.'s definition of an attack pattern) and execution flows (element 3 in Moore et al.'s definition of an attack pattern) for each CAPEC in *SC*. This combination is then converted into sentence tokens using the Natural Language Toolkit (NLTK) Python library [8] and preprocessed to retain only alphanumeric characters and spaces. Similarly,

[6] https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/tree/master/capectracer?ref_type=heads
[7] https://capec.mitre.org/data/downloads.html

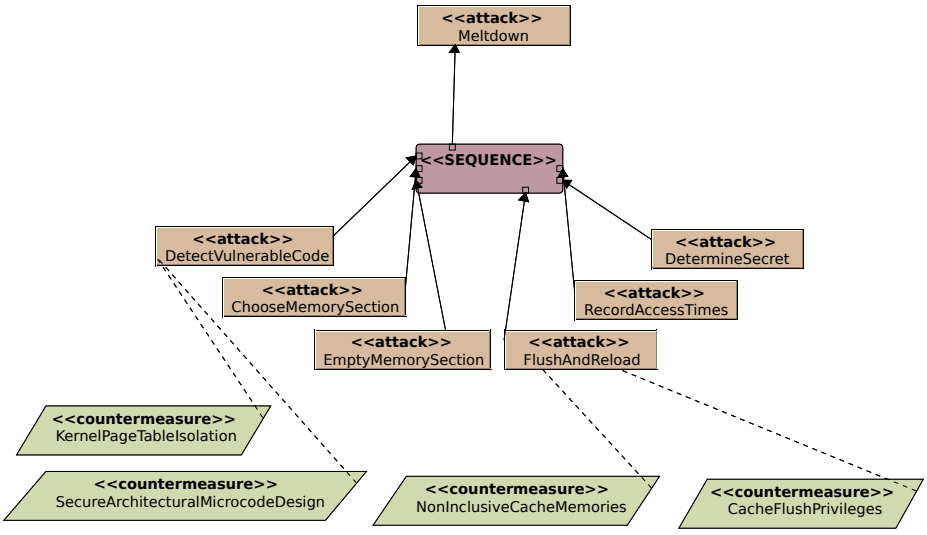

Figure 3: Screenshot showing example branches of an ADTGC CPU ADT

the sentences in the leaf node description are tokenized with NLTK and undergo the same preprocessing as the CAPEC sentences.

Each leaf node description and CAPEC sentence is converted into sentence embeddings, using a pretrained Sentence Transformer model called ATT&CK BERT [6, 18], which was initially developed to map CVEs to MITRE ATT&CK techniques[8]. We denote as *SELN* (resp. *SEC*) the set of sentence embeddings for a leaf node description (resp. the set of sentence embeddings corresponding to a CAPEC sentence).

The CAPEC Tracer subsequently assesses the relevance of each CAPEC in *SC* to every leaf node description. This is achieved by constructing a two-dimensional matrix of cosine similarity scores, referred to as *CM*. Each element in *CM* corresponds to a cosine similarity score between a sentence embedding from *SELN* and a sentence embedding from *SEC*. Thus, *CM* gathers the cosine similarity scores for all possible pairs of sentence embeddings between *SELN* and *SEC*. Afterwards, the arithmetic mean is calculated for all the cosine scores in *CM* to produce an average cosine similarity score. This process is repeated for each CAPEC in *SC*, resulting in a sorted list of indexed scores. The list is then filtered to remove scores below 0.4 and those associated with CAPECs lacking mitigation data. Finally, the indexes from this filtered list are used to extract and return the mitigation data from their corresponding CAPECs.

4.3.2 ADT Generator. The second sub-mechanism of *ADTGC* is the ADT Generator, that is responsible for interacting with the LLM to produce a list of pertinent countermeasures. This generation process is carried out as described in stages 3 and 4.

It relies on the interaction mechanism between TTool and OpenAI's API provided by TTool-AI [3], and the automatic generation and parsing of successive prompts, adapting the TTool-AI core mechanisms (contextual knowledge injection, JSON format, automated feedback loop) to meet the requirements for ADT generation. More in detail, our tool sends first a prompt "learning" to the LLM the expected response format:

```
When you are asked to identify mitigations for a
    provided list of attack steps, return them as a
    JSON specification formatted as follows:
"mitigations": [{\"name\": \"NameOfMitigation\", \"
    description\": \"The description of the
    mitigation and how it prevents the attack.\"}, \"
    attacksteps\": [\"TheNameOfAProvidedAttackStep\"
    ...]} ...]}
# Respect: All words in \"name\" must be conjoined
    together.
# Respect: There must be no more than forty
    characters in \"name\".
# Respect: For each word in \"name\", its first
    letter must be capitalized.
[...]
```

Following this, the tool sends another prompt with descriptions of the attack steps and potential countermeasures identified by the CAPEC tracer. It then injects a prompt tasking the LLM with countermeasure identification and pairing:

[8]https://attack.mitre.org/

```
Using the specified JSON format, identify a list of
    possible mitigations that would prevent an
    attacker from using most, if not all, of the
    provided attack steps to further advance the
    attacker's attack scenario. Each mitigation
    should be associated with at least one attack
    step from the provided attack step list. If
    applicable, use the provided list of possible
    countermeasures as support for identifying
    mitigations. Do respect the JSON format, and
    provide only JSON (no explanation before or after
    ).
```

The LLM's response is then parsed. If there is a syntax error in the JSON structure, or if the JSON is empty, the following prompt is generated, containing the detected errors:

```
Your answer was as follows: [LLM Response]
Yet, it was not correct because of the following
    errors:
[Errors description]
```

This prompt is then sent to the LLM, and the next response is parsed. This process is repeated until no errors are detected or after ten consecutive reworked responses are received. Once the final response from the LLM is received and the user is satisfied with it, they click the "Apply Response" button in the TTool-AI GUI. This action updates the attack tree, converting it into an ADT.

5 EVALUATION

5.1 Evaluation Methodology

To evaluate our contribution, we first selected three input attack tree diagrams. The attack trees were created using TTool-AI attack tree generator [1], based on three input system specifications. These specifications described three distinct systems: a cloud service infrastructure, a social network mobile application, and a CPU. From there, a security engineer with two years of professional experience was selected to manually create ADT diagrams from the aforementioned attack tree diagrams. This included identifying as many relevant countermeasures as possible and mapping this identified set with appropriate attack leaf nodes. Part of this task also included providing titles and descriptions for each countermeasure. The engineer was given an hour maximum to complete this task and diagram the results as a separate tree.

For each attack tree, we also tasked both versions of our methodology (*ADTGC* and *ADTG*) to generate an ADT. Thus, a total of nine ADT diagrams were created among *ADTGC*, *ADTG*, and the engineer. Figure 3 shows example branches of an *ADTGC*-generated ADT.[9]

The ADTs were then evaluated using the metrics outlined in Section 4.1, those created by hand being evaluated by two others security experts. For determining the *MM* score in each ADT, the evaluators had a time limit of thirty minutes to identify additional countermeasures and pair them with attack leaf nodes. All other metrics were assessed without any time constraints.

[9]Note that due to space constraints, this screenshot only shows a sample of identified countermeasures while in the actual generated diagram there are additional connected countermeasures.

5.2 Results

The grading of the ADT is summarized in Table 1. We can make from these results the following observations:

- Both *ADTGC* and *ADTG* generate countermeasures significantly faster than the manual methodology, with *ADTG* being particularly efficient.
- The manual methodology produced better detailed countermeasures and more relevant countermeasure pairings compared to *ADTGC* and *ADTG* when comparing their average semantic correctness scores. However, *ADTGC* produced much better results from this perspective than *ADTG*, especially when it came to the CPU ADT that contained low-level and complex attack nodes.
- The manual and *ADTGC* methodologies produced sets of countermeasure pairings with less omissions from these sets compared to *ADTG* as seen with their average completeness scores. In addition, *ADTGC* was not that far off from the manual methodology when it came to this perspective as *ADTGC* was able to identify at least commonly recognized countermeasures for each attack tree.

From these observations, it can be inferred that the only advantage of creating ADTs with *ADTG* is the speed at which it does so. Otherwise, *ADTGC* on average produces better ADTs than *ADTG* and also produces these trees at a quality not that far off from a professional security engineer at the associate level, especially in regards to their average completeness scores only having a ~19% difference. This is also evidenced by *ADTGC* having fairly low standard deviations for its averaged scores — 0.08 for example in regards to the standard deviation of its average completeness score. Therefore, we have decided to retain *ADTGC* for inclusion in TTool's release.

5.3 Threats to Validity

Even though this experimentation lays a foundation for evaluating our approach, it has several limitations that may affect its validity. Firstly, the number of input attack trees is limited, with only three attack trees serving as the basis for the experimentation. Additionally, regarding the evaluation: the manual ADTs were evaluated by two security experts, whereas the AI-generated ADTs were evaluated by the same security expert who designed the manual ADTs. This introduces unavoidable evaluator variability, although we used well-defined metrics. Nonetheless, this variability is somewhat constrained by the fact that several scores in the metrics are objective.

6 CONCLUSIONS

Complementing TTool-AI's attack tree generation feature [1], this paper presents an extension to this prior approach by incorporating mitigations into existing attack trees. Our contribution relies on LLMs enriched with additional cybersecurity context extracted from the CAPEC database. This is achieved using a CAPEC Tracer, which employs NLP techniques to identify and select relevant CAPECs based on the attack leaves in the input trees. Evaluation results

Table 1: Evaluation results

(a) Cloud service case study

Creator	Complexity	Semantic correctness	Completeness	Generation time (s)
Engineer	8	1.00	0.42	3600
ADTG	9	0.67	0.35	34
ADTGC	10	0.70	0.37	633

(b) CPU case study

Creator	Complexity	Semantic correctness	Completeness	Generation time (s)
Engineer	5	1.00	0.50	3600
ADTG	11	0.09	0.09	23
ADTGC	9	0.44	0.31	882

(c) Social network application case study

Creator	Complexity	Semantic correctness	Completeness	Generation time (s)
Engineer	6	1.00	0.46	3600
ADTG	15	0.40	0.40	23
ADTGC	17	0.41	0.47	912

(d) Statistics

Creator		Complexity	Semantic correctness	Completeness	Generation time (s)
Engineer	Average	6.33	1.00	0.46	3600
	Std. Dev.	1.53	—	0.04	—
ADTG	Average	11.67	0.39	0.28	26.67
	Std. Dev.	3.06	0.29	0.17	6.35
ADTGC	Average	12	0.52	0.38	809
	Std. Dev.	4.36	0.16	0.08	153.16

demonstrate that while our method does not yet produce ADTs as complete and semantically accurate as those created by security experts, it serves as a valuable basis for security analysis by delivering high-quality results efficiently.

To enhance the completeness and semantic accuracy of the generated attack/countermeasure pairings, future work will focus on several key areas: integrating other LLMs into our implementation, refining our CAPEC Tracer to improve the computation of similarity scores for more precise mitigation selection, incorporating databases beyond CAPEC to broaden the mitigation basis, and conducting further prompt engineering to optimize the prompts automatically sent to the LLM by our tool.

In addition, currently our methodology is only capable of creating ADTs from attack-trees structured under Kordy et al.'s standard format [15]. However, attack trees are oftentimes structured differently such as the attack trees used by Wang and Liu [27] which contain node representations of 'Actions', 'Detections', and 'Deceptions' in addition to traditional attack nodes. As such, our methodology could be expanded to additionally convert attack trees with different formats from Kordy et al.'s and also to account for all attack nodes, not just leaf attack nodes, without providing excess countermeasures and countermeasure pairings.

Finally, our evaluation approach has several limitations, and our contribution requires complementary assessments for a more comprehensive evaluation. A comparative study with other available generation methods would be highly beneficial to determine its relative impact with respect to the existing literature.

REFERENCES

[1] Alan Birchler De Allende, Bastien Sultan, and Ludovic Apvrille. 2024. Automated Attack Tree Generation Using Artificial Intelligence & Natural Language Processing. Proceedings of the 19th International Conference on Risks and Security of Internet and Systems (CRISIS 2024) (TO APPEAR).

[2] Ludovic Apvrille and Bastien Sultan. 2024. AI-Driven Consistency of SysML Diagrams. In *Proceedings of the ACM / IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS'2024), Foundations Track, Linz, Autria, Sept. 2024 (TO APPEAR).*

[3] Ludovic Apvrille and Bastien Sultan. 2024. System Architects are not Alone Anymore: Automatic System Modeling with AI. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering - MODELSWARD - BEST PAPER AWARD.* INSTICC, SciTePress, 27–38. https://doi.org/10.5220/0012320100003645

[4] Georgios Bakirtzis, Bryan T Carter, Carl R Elks, and Cody H Fleming. 2018. A model-based approach to security analysis for cyber-physical systems. In *2018 Annual IEEE International Systems Conference (SysCon).* IEEE, 1–8.

[5] Sean Barnum. 2008. Common Attack Pattern Enumeration and Classification (CAPEC) Schema Description. Cigital.

[6] Abdeen Basel, Ehab Al-Sheer, Anoop Singhal, Latifur Khan, and Kevin Hamlen. 2023. SMET: Semantic Mapping of CVE to ATT&CK and its Application to Cyber Security. DBSec 2023: Data and Applications Security and Privacy XXXVII, Sophia Antopolis, FR. https://doi.org/10.1007/978-3-031-37586-6_15

[7] Sahar Berro, Ludovic Apvrille, and Guillaume Duc. 2019. Optimizing System Architecture Cost and Security Countermeasures. In *International Workshop on*

Graphical Models for Security. Springer, 50–67.

[8] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[9] Binbin Chen, Christoph Schmittner, Zhendong Ma, William G Temple, Xinshu Dong, Douglas L Jones, and William H Sanders. 2015. Security analysis of urban railway systems: the need for a cyber-physical perspective. In *Computer Safety, Reliability, and Security: SAFECOMP 2015 Workshops, ASSURE, DECSoS. ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings 34*. Springer, 277–290.

[10] Sophie Coudert, Ludovic Apvrille, Bastien Sultan, Onana Hotescu, and Pierre de Saqui-Sannes. 2024. Incremental and Formal Verification of SysML Models. *Springer Nature Computer Science* (2024).

[11] Kenneth Edge, Richard Raines, Michael Grimaila, Rusty Baldwin, Robert Bennington, and Christopher Reuter. 2007. 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)*. IEEE, 144b–144b.

[12] Simon Yusuf Enoch, Jang Se Lee, and Dong Seong Kim. 2021. Novel security models, metrics and security assessment for maritime vessel networks. *Computer Networks* 189 (2021), 107934.

[13] Barbara Fila and Wojciech Wideł. 2020. Exploiting attack–defense trees to find an optimal set of countermeasures. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 395–410.

[14] Florian Kammüller. 2019. Combining secure system design with risk assessment for IOT healthcare systems. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 961–966.

[15] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. 2014. Attack-defense trees. *Journal of Logic and Computation* 24 (02 2014). https://doi.org/10.1093/logcom/exs029

[16] Dmitry Levshun, Yurii Bakhtin, Andrey Chechulin, and Igor Kotenko. 2019. Analysis of attack actions on the railway infrastructure based on the integrated model. In *International Symposium on Mobile Internet Security*. Springer, 145–162.

[17] Andrew Moore, Robert Ellison, and Richard Linger. 2001. *Attack Modeling for Information Security and Survivability*. Technical Report CMU/SEI-2001-TN-001. https://doi.org/10.1184/R1/6572063.v1

[18] Nils Reimers and Iryna Gurevych. 2019. 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. http://arxiv.org/abs/1908.10084

[19] Yves Roudier and Ludovic Apvrille. 2015. 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)*. INSTICC, 655–664.

[20] Arpan Roy. 2010. Attack countermeasure trees: A non-state-space approach towards analyzing security and finding optimal countermeasure sets.

[21] Arpan Roy, Dong Seong Kim, and Kishor S Trivedi. 2012. Scalable optimal countermeasure selection using implicit enumeration on attack countermeasure trees. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.

[22] Bruce Schneier. 1999. Attack trees. *Dr. Dobb's journal* 24, 12 (1999), 21–29.

[23] Muhammad Ali Siddiqi, Robert M Seepers, Mohammad Hamad, Vassilis Prevelakis, and Christos Strydis. 2018. Attack-tree-based Threat Modeling of Medical Implants.. In *PROOFS*. 32–49.

[24] Orly Stan, Ron Bitton, Michal Ezrets, Moran Dadon, Masaki Inokuchi, Yoshinobu Ohta, Tomohiko Yagyu, Yuval Elovici, and Asaf Shabtai. 2021. Heuristic Approach for Countermeasure Selection Using Attack Graphs. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. https://doi.org/10.1109/CSF51468.2021.00003

[25] Bastien Sultan, Ludovic Apvrille, Philippe Jaillon, and Sophie Coudert. 2023. W-Sec: A Model-Based Formal Method for Assessing the Impacts of Security Countermeasures. In *Model-Driven Engineering and Software Development*, Luís Ferreira Pires, Slimane Hammoudi, and Edwin Seidewitz (Eds.). Springer Nature Switzerland, Cham, 203–229.

[26] Bastien Sultan, Léon Frénot, Ludovic Apvrille, Philippe Jaillon, and Sophie Coudert. 2023. AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models. *ACM Trans. Embed. Comput. Syst.* (sep 2023). https://doi.org/10.1145/3624583

[27] Ping Wang and Jia-Chi Liu. 2014. Threat Analysis of Cyber Attacks with Attack Tree+. *J. Inf. Hiding Multim. Signal Process.* 5, 4 (2014), 778–788.