information

Article

Formal Template-Based Generation of Attack–Defence Trees for Automated Security Analysis

Jeremy Bryans [1,*], Lin Shen Liew [2], Hoang Nga Nguyen [3], Giedre Sabaliauskaite [3] and Siraj Ahmed Shaikh [3]

1 Systems Security Group, Centre for Future Transport and Cities, Coventry University, Coventry CV1 5FB, UK
2 iTrust, Singapore University of Technology and Design, Singapore 487372, Singapore; linshen0502@gmail.com
3 Systems Security Group, Department of Computer Science, Swansea University, Swansea SA1 8EN, UK; h.n.nguyen@swansea.ac.uk (H.N.N.); g.sabaliauskaite@swansea.ac.uk (G.S.); s.a.shaikh@swansea.ac.uk (S.A.S.)
* Correspondence: ac1126@coventry.ac.uk

Abstract: Systems that integrate cyber and physical aspects to create cyber-physical systems (CPS) are becoming increasingly complex, but demonstrating the security of CPS is hard and security is frequently compromised. These compromises can lead to safety failures, putting lives at risk. Attack Defense Trees with sequential conjunction (ADS) are an approach to identifying attacks on a system and identifying the interaction between attacks and the defenses that are present within the CPS. We present a semantic model for ADS and propose a methodology for generating ADS automatically. The methodology takes as input a CPS system model and a library of templates of attacks and defenses. We demonstrate and validate the effectiveness of the ADS generation methodology using an example from the automotive domain.

Keywords: automotive cyber security; threat modelling; attack defense tree

check for updates

Citation: Bryans, J.; Liew, L.S.; Nguyen, H.N.; Sabaliauskaite, G.; Shaikh, S.A. Formal Template-Based Generation of Attack–Defence Trees for Automated Security Analysis. *Information* **2023**, *14*, 481. https://doi.org/10.3390/info14090481

Academic Editor: Sokratis Katsikas

Received: 14 June 2023
Revised: 21 July 2023
Accepted: 22 July 2023
Published: 29 August 2023

1. Introduction

Cyber Physical Systems (CPS) are complex systems, which integrate digital capabilities (networking, data, computational systems) with physical devices and engineering systems. Their application domains include transportation, energy, healthcare, manufacturing, agriculture and other sectors. They are expected to enhance performance, increase efficiency and safety, and to improve quality of life. However, to meet these expectations, CPS themselves must be safe and secure.

Attack Trees (AT) have been one of the most used tools for security analysis, probably due to its graphical and tree-like notation that can succinctly describe a set of possible ways (aka paths) of realizing a complex realization of a threat scenario (aka attack). Basically, an AT [1] has one root node denoting the attacker's goal; every node may have multiple child nodes denoting its refinement into sub-goals; those without child nodes are regarded as leaf nodes representing the attacker's basic actions; a refinement can be disjunctive (OR) or conjunctive (AND); such hierarchical structure of AT enables one to visualize hence easily grasp how attacks can succeed by progressively achieving certain basic actions and sub-goals in series and/or parallel. Besides, it also allows qualitative/quantitative analysis—for instance, a bottom-up computation of attributes such as attack likelihood/potential and time/cost required.

ATs can be used for determining security risk level of various threat scenarios. Recent international standard for road vehicle cybersecurity, ISO/SAE 21434 [2], proposes to use attack tree analysis as one of methods for deciding threat feasibility level. Then, cybersecurity risk level is computed by combining threat feasibility and impact levels. Depending on the computed risk level, suitable countermeasures can be selected to mitigate the risk.

To further enhance AT's expressiveness and capabilities, various extensions have been proposed in the literature over the last 20 years and they may be diverse in terms of application domain, interpretation, formalism, etc. [3,4]. One of the notable extensions is the introduction of sequential conjunction (SAND). It's intended primarily to express the order in which the attack steps (i.e., child nodes of a given node) should be achieved, which the classical AT [1] is incapable of; additionally, SAND could also be used to express temporal dependencies between nodes [3,5]. A rigorous mathematical formalization of ATs with SAND is given in [6]. There are other efforts that express sequential relation among the nodes of AT without using SAND. For instance, refs. [7,8] assume that the children of AND-node are achieved in sequential order from left to right. Besides, ref. [9] uses dotted arrow lines to link the relevant child nodes of any parent nodes; this however may result in multiple arrow lines intersecting/overlapping the AT, thereby deteriorating its readability. Appropriate (and non-excessive) use of additional symbols (e.g., SAND operator) could help improve the expressiveness on ATs without compromising their readability.

Additional processes/steps/nodes denoting mitigation measures/strategies have been integrated into ATs to reason/model the interactions between attacker and defender, thereby leading to various extended ATs named Defense Tree [10–12], Countermeasure Graph [13], Bounded Decision Markov Processes [9], Attack Countermeasure Tree [14], and Attack Defense Tree [3,15], just to name a few. Such extensions are largely motivated by the pressing need for evaluating how effective are the selected measures in detecting/preventing/mitigating the attacks or whether the potential risks are reduced to acceptable levels.

The process of constructing attack and attack defense trees is time-consuming. Furthermore, the analysis comprehensiveness is largely dependent on how accurately/precisely the tree (which the analyst constructs) can model the analyzed system. Simply put, it requires the analyst to possess sufficient expertise and knowledge of the analyzed system to produce the tree that correctly models the system to ensure effective analysis. In addition, as the size of AT generally grows with the system architectural complexity, the detailed ATs for modern complex systems can grow too large and thus difficult for human analysts to follow or estimate particular attributes such as attack likelihood. Moreover, manual construction of a large AT is likely to be labor intensive and prone to human error. Duplicated nodes are not rare especially for large systems, yet the same analyst might carelessly refine them into inconsistent combinations of child nodes. In addition, ATs constructed by individual analysts for the same system may differ considerably in terms of structure and size [3,16].

The aforesaid limitations of manual construction of AT have thus given rise to automated AT generation methods, such as [17–19]. This paper proposes a template-based method for automated generation of attack defense trees with sequential conjunction to facilitate CPS cybersecurity analysis. To the best of our knowledge, this has not been done before. The paper brings together the following contributions:

1. a formal representation of the ADS tree (attack defense tree with sequential conjunction), followed by a demonstration that this representation is equivalent to a process-algebraic one;
2. an algorithm that checks the ADS tree and tells whether the attack or defense is successful;
3. a methodology for generating ADS trees based on two inputs: (i) a simple network model depicting the hardware components involved in the system of interest; (ii) a library consisting of component-specific templates as well as documented/recommended/applicable measures/defenses.

The first of these contributions is a alternative semantics for the ADS that allows the use of model-checking to reason about the ADS. It is equivalent to the semantics from [6]. In this paper we recall the semantics from [6] and the and new, equivalent semantics (previously presented in [20]). This positions us to develop the second contribution—the algorithm that allows us to identify whether attacks will be or will not be successful. The third point (the methodology) uses the reasoning algorithm to check for undefended attacks. Together, these points show a way in which vehicle manufacturers can develop and keep security cases up-to-date during the lifetime of the vehicle. Security cases are a requirement of the new automotive security standard ISO21434 [2].

The remainder of the paper is organized as follows. Section 2 includes a review of related literature. Background information is described in Section 3. Section 4 defines syntax and semantics, while Section 5 explains reasoning about ADS trees. A methodology for automated ADS tree generation is presented in Section 6. Finally, Section 7 concludes the paper and describes directions for future work.

2. Literature Review

Attack trees (AT) and their derivatives have been widely used for handling threat modelling and security risk assessment, as they can succinctly describe complex threat scenarios and help their readers (e.g., analysts) to better observe and reason about the impact of risks on the system. However, manual construction of AT is heavily dependent on one's expertise and experience (ATs constructed by individual analysts for the same system can differ structurally [3,16]). Experts often use libraries of common attack patterns or reuse parts of models. This can render reading and maintaining ATs even more challenging for third parties. Besides, Attack Trees can also be verbose and error-prone, especially when the size of the AT and its corresponding system complexity becomes substantial. A viable solution to these issues, and one which we explore in this paper, is automated Attack Tree generation. This is considered in the table (Table 1). Another theme we explore in this literature survey is the validation of Attack Trees.

A significant survey was published by Wideł et al. [21] in 2020 and covers the period 2014–2019. It has a focus on applications in formal methods for attack tree based security modeling, dividing the surveyed work into strict semantical work, work on the generation of attack trees and quantitative approaches for attack trees. Our interest, and our approach here, is the semantic basis for the generation of attack (attack defense) trees. In [21] the authors observe that manually constructed AT are subjective and AT describing the same system can differ widely in size, structure and even the attack vectors captured. The authors also discuss the more expressive semantics given in [22], in which AND and SAND operators are distinguished using a logical semantics. Audinot's work on path interpretation [23,24], has the goal of using the path interpretation framework to allows a user to express and check whether an attack tree is consistent with the analyzed system, and is the basis for the ATSyRA tool [25].

Pinchinat et al. [19] also uses the path interpretation semantics, and consider the problem of access to physical locations, using a version of the tool ATSyRA developed using the Eclipse Modelling Framework [25]. ATSyRA then compiles this into a symbolic transition system. The authors observe that "naive fully automated generation is likely to produce unexploitable trees (because they are flat)" and consequentially they synthesise structure into their attack tree using factorisation, with the structuring rules necessary provided by experts. In our work structure is provided through templates (patterns) built into attack trees derived from the analysed system as the attack tree is being created.

Vigo, Neilsen and Neilsen [26,27] use attack trees to communicate security information in a structural and succinct way to non-experts in a company. They tackle the generation problem with a system modeled using a variant of the π calculus (a value-passing quality calculus) and a target location or asset, and generate an attack tree showing how an attack on the system may be carried out. In their specification, the attacker model is an adversary process able to obtain cryptographic access to any channel. These channels may be cyber (e.g., cryptography) or physical (e.g., reinforced gates) protection mechanisms. Additionally, if a map from channels to costs is provided, the proposed framework can compute the cheapest sets of channels that enable attaining a location of interest. The system used in their paper is a communication system (NEMID—Danish for EASYID) as the target system. The value-passing calculus allows to model scenarios beyond the standard network security domain, and enables designing syntax-directed static analysis, avoiding the state space explosion suffered by model checking algorithms. Our work differs from this in its targetted application to automotive networks and it's use of the process algebra CSP. Our atttacker model is similar to [26], where the authors have the implicit assumption that the attacker is capable of obtaining the necessary access at each step. The use of the templates assumes the at each step the access required. The network architecture we use would be available as part of the vehicle development process, and taking it as an input here means that our method will be usful to the vehicle development process. [27] look at developing Attack Trees and quantifying the protection of the system modelled. The approach uses the same syntax-directed approach as previously, but uses a different process algebra (although from the π-calculus family—the protection calculus [28]). They use this to consider the cost of an attack, an area we do not touch on in this paper.

Ivanova et al. [7] use recursive policy invalidation to generate an attack tree showing how a socio-technical system can be attacked. They generate Attack Trees based on Directed Graphs. The methodology specifically aims to take "human factor" issues into account, and is demonstrated with an example of a socio-technical system modelled using a (directed) graph with nodes representing locations (e.g., rooms and computers), actors (e.g., persons), processes (modelling information sharing or policies), and items (tangible assets like access cards and hard-drives). It forms part of the TREsPASS approach [29]. The TREsPASS model includes security controls such as access control policies that limit access to certain locations and digital means of enforcing these (for example password mechanisms). The socio-technical aspect is absent in our work, but would be a worthwhile addition.

A few papers consider the issue of validation of attack trees. In [18] the method is for entirely new systems, rather than for existing ones. The validation focusses on usability and focusses on issues such as Attack Tree formatting for the end-user, and the length of the descriptive names used for Attack Tree nodes. It is designed for operational processes rather than architectures than include communications between components.

In the automotive domain, there have been several efforts in proposing systematic approaches to constructing attack trees. Paul's methodology [18] for generating Attack Trees is demonstrated with an example of automotive braking system. The methodology she presents requires three inputs, namely the system's logical and operation architectures modelled via a particular architecture framework by Thales Melody, the system's risk assessment result, and a security knowledge database (e.g., EBIOS). Using these inputs, a methodology to produce an Attack Tree is then given. She proposes a systematic approach based on these inputs to automate the construction and maintenance of attack trees. Similarly, an initial work by Kacper et al has been proposed in [30] to create an automated Machine Learning-based framework to construct attack trees with the main motivation for application in automotive cybersecurity. The approach utilises Graphical Neural Network to enable the automated construction with training samples are extracted from existing vulnerability databases. In contrast to this, Mahmood et al. [31] proposed a structural approach to construction of attack tree for automotive Over The Air Update systems. Their work suggests to first perform a threat analysis from inputs such as structure models of these systems. Although attack tree outputs from this approach is tracable to elements in the input, its construction is currently manual and requires experts' knowledge and experience.

Security evaluation is discussed by Cheah et al. [32], but there the purpose of the evaluation is to evaluate severity ratings with respect to discovered weaknesses. The evaluation is specifically "from a black box perspective", and it is assumed that components are supplied by third-parties, and the evaluation is carried out by domain experts. Attack trees are used to map the process of penetration testing that is automatically out. Lallie et al. studies the question of cyber attack perception in [33] and evaluate two different attack modelling techniques (including attack trees) with respect to their ability to communicate attacks to different groups of people. Importantly, though, the focus is on the attack, rather than the correctness of the illustration of the attack.

Hong et al. [34] attempt to construct ATs automatically while maintaining all possible attack scenarios. They propose two logic reduction techniques that automatically generate optimally structured AT that are smaller in size yet enabling attack cost estimation equivalent to that of their original full forms, which would usually contain repetitive or superfluous branching. The evaluation is to confirm that the simplified ATs provide the same security analysis compared to the full AT. An example security analysis is used as a means for all the ATs produced.

Gadyatskaya et al. [15] present an approach to adding defenses to AT. The system is transformed into a Directed Graph following the TREsPASS approach presented in [7]. A method using "attack-defense bundles" is proposed by making explicit in the attack generation process the fact that the attacker needs to overcome the restrictions imposed by security policies. The focus of the authors is on socio-technical systems. Each identified asset (of the resultant Directed Graph) results in a (small) attack defense tree which is derived from a template (a bundle). This template consists of 7 types of nodes—5 types for attack and 2 types of defense. So, if multiple assets are accessed by the attacker in parallel and/or series to reach the target location/asset, then the accessed assets' corresponding (small) attack defense tree are merged to form a (big) attack defense tree with OR and AND refinement. Note that the method is not fully formal, because the "defense" nodes which are subsequently added to the ADT are expressed in the natural language. These defense-nodes can denote preventive, detective or corrective controls. The choice of defenses (controls) is up to the human analyst.

In [35] Gadyatskaya et al. present another approach to generating Attack Trees. They use the SP (series-parallel) semantics to present a theoretical framework that addresses the problem of attack tree generation, deriving the refinement structure from an abstraction relation on system predicates. The approach, unlike the aforementioned ones, does not identify the attack paths (traces). Given a set of attack paths and a set of refinement rules, the approach would output an optimized and correct AT through a greedy heuristic based on the edge biclique problem. They aim to construct the tree such that it contains structure—non-leaf nodes are semantically meaningful, and give a heuristic algorithm to solve this. Validation of the proposed technique based on realistic case studies is left to further work.

Chlup et al. [36] present an approach to automated construction of attack trees using the notion of an anti-pattern, which is essentially a "pattern that describes a configuration that cyber-resilient systems should not include (i.e., a vulnerability)". Using a separate system model and threat model, attack paths can be constructed. Ultimately, the depth and breadth of attack trees rely on the combination of the initial set of anti-patterns (subject to the nature of components used in the system) and attacker capabilities (subject to any assumptions defined by the user). While the approach is well implemented—widely known as the THREATGET tool—there is limited validation on effective and complete are the attack trees.

Gadyatskaya and Mauw (in [37]) present an interesting change in point-of-view, moving from a static view of single stand-alone attack trees to a time-indexed set and thus introducing the possibility of dynamic security monitoring. The authors make the case that dealing with the series allows the detection of trends in dynamic security scenarios on attack trees. The model (Attack Tree Series) also allows the visualization of the evolution of the security posture with respect to the considered threat model. Attack Tree Series also allow quantitative threat data values to change over time, and capture the importance of these changes in a *temperature* function. A temperature function is a history-dependent attribute, meaning that it is based on the history of values of an attribute. Tracking changes in this temperature function enables the analysis and monitoring of historical trends. The work is validated using an example from teller machines fraud.

An alternative approach to adding time to an attack tree is taken by Ali and Gruska in [38]. The authors allow (min,max) time constraints to be associated with attack nodes.

The full attack tree can then be analysed if the attack node occurs at any time within the associated interval. The timed automata tool UPPAAL can then be used for analysis.

Jhawar et al. in [39] present a method aimed at re-use and based on annotating AT with assumption and guarantee predicates that indicate how they may be combined. They then build attack trees using libraries of patterns drawn from vulnerability databases. This then provides the automated step of a semi-automatic Attack Tree curation process. The work is validated by using a library of attack trees generated from standardised vulnerability descriptions in the National Vulnerability Database, then using trees from this library to augment a manually constructed annotated attack tree representing a high level attack pattern described in Mitre's Common Attack Pattern Enumerations and Classifications [40].

Attack Countermeasure Trees (ACT) are presented in Roy et al. [14], where countermeasures including detection and mitigation are allowed at any level of the tree. The purpose of the work is to aid in the development of a scheme to identify where in the system security investment should be prioritized. The work relies on several probabilistic analyses to compute the impact and cost of a successful attack as well as the system's risk to a particular attack scenario. The paper provides an input tool in order to evaluate an attack countermeasure tree in terms of impact and cost.

There are further two contributions to semantics that are noteworthy. In the first [41] Mantel et al. argued that AT are not "explicit" because the trees do not clarify the connection between the attacks and the attacker goal of the tree explicitly. The authors present a framework in which the relationship of the attacks to the attackers goal at the top of the tree can be made explicit. The work is closely related to the SP semantics. The authors observe that the success of an attack depends on (i) the actions of the attacked system, (ii) the actions of the attacker, (iii) possibly the actions of other actors, and the interplay between these factors. They then identify three degrees of freedom in the specification of the success of an attack: purity (may occurrences of actions of an attack be interleaved with occurrences of other actions and, if yes, which ones?), persistence (is it sufficient if the attacker's goal is satisfied at some point in time or should it be satisfied persistently?) and causality (can we be certain that the goal is a result of the attack?). They develop a trace-based language to describe the behaviour of attackers and attacked systems, allowing the question "Does an attack achieve an attacker goal in a run?" to be answered using their three-element framework. They then give various ways of answering these questions to describe different points on the framework. Finally, they evaluate selected papers from diverse domains with respect to the persistence, purity and causality, showing that the attack trees from quite diverse domains can have their success criteria fitted into the same framework.

In a similar piece of work to [41], Pinchinat et al. in [42] extend the model of an AT with a model of the analysed system. This allows the authors to take the specific system configuration into account, and to be precise about the path semantics of the attacks. The use of the analysed system allows them to identify which attacks described by the attack tree are possible in the system, arguing that generic attack trees may contain attacks which are impossible to execute on a system of interest. In this work we take a similar approach to [42]. We use a system model to derive attacks, and apply it also to defenses to get the attack defense tree. We also use templates of known attacks and defenses that are specific to the automotive world.

The previous work on which this paper is built is related to both [41] (in our use of the SP semantics) and [42] (in our use of the system model). In [20] we set up a semantic framework, and in [43] we derive an Attack Tree from a system model and a library of attack patterns. In the current paper we extend this work to Attack and Defense patterns, and a new methodology is given that generates an Attack Defense Tree (as an ADS).

Table 1. A survey of papers tackling automatic generation of Attack Trees. (AT: Attack Trees; AG: Attack Graphs; ACT: Attack Countermeasure Tree; ADT: Attack Defence Tree; ADS: Attack Defence Tree with sequential conjunction; OR/AND—standard Boolean operators; SAND—sequential AND).

Algorithm	**Automated Generation**	**Output Model**	**Operators**	**Running Example**
[44,45]	Yes	AG	-	Computer system
[34]	No	AT	OR/AND	Random network system
[18,30]	Yes	AT	OR/AND	Automotive system
[31]	No	AT	OR/AND/SAND	Automotive system
[17,19]	Yes	AT	OR/AND/SAND	Military building
[7,46]	Yes	AT	OR/AND	Socio-technical system
[26,27]	Yes	AT	OR/AND	National authentication system
[15]	Semi	ADT	OR/AND	Socio-technical system
[47]	No	AT	OR/AND	Medical system
[48]	Semi	ACT	OR/AND	Computer system
[35]	Yes	AT	OR/SAND	Computer system
[39]	Semi	AT	OR/AND	Computer system
[24]	No	AT	OR/AND/SAND	Military building
[49]	Yes	AT	OR/AND	Urban surveillance system
[37]	No	AT	OR/AND	ATM system
[36]	Yes	AT	OR/AND	Automotive system
[43]	Yes	AT	OR/AND/SAND	Automotive system
Current Paper	Yes	ADS	OR/AND/SAND	Automotive system

Table 1 summarizes differences among works by assessing their algorithms with respect to the following questions:

- Is the output model produced in the cited work automatically generated?
- What graphical model does the method output? These can be Attack Trees, Attack Defense Trees, Attack Defence Trees with sequential conjunction; Attack Counter Measure Trees or Attack Graphs?
- Which operators does the output model use? These are OR, AND and SAND operators.
- What domain inspired the running example provided to demonstrate the application of the method discussed?

3. Background

3.1. Attack Trees

Attack trees contain a set of leaf nodes, structured using the operators conjunction (**AND**) and disjunction (**OR**). The leaf nodes represent atomic attacker actions. The **AND** nodes (resp. **OR** nodes) are complete when all child nodes have (resp. at least one child node has) been carried out.

Extensions have been proposed using **Sequential AND** (or **SAND**) [6]. We follow the formalisation of attack trees given in [6,16]. If $\mathbb{A}$ is the set of all possible atomic attacker actions, the elements of the attack tree $\mathbb{T}$ are $\mathbb{A} \cup \{\mathbf{OR}, \mathbf{AND}, \mathbf{SAND}\}$, and an attack tree is generated by the following grammar, where $a \in \mathbb{A}$:

$$t ::= a \mid \mathbf{OR}(t, \ldots, t) \mid \mathbf{AND}(t, \ldots, t) \mid \mathbf{SAND}(t, \ldots, t)$$

Attack tree semantics have been defined by interpreting the attack tree as a set of series-parallel (SP) graphs [6]. We use the definition of SP graphs. It first requires the definition of source-sink graphs and here we use the definitions from [6].

Definition 1. *A source-sink graph over $\mathbb{A}$ is a tuple $G = (V, E, s, z)$ where V is a set of vertices, E is a multiset of edges with support $E* \subseteq V \times \mathbb{A} \times V$, $s \in V$ is a unique source and $z \in V$ is a unique sink, and $s \neq z$.*

Two example source sink graphs are given in Figure 1. The sequential composition of G and another graph G', denoted by $G \cdot G'$ results from the disjoint union of G and G'

and linking the sink of G with the source of G' (linking z with s'). Thus, if $\dot{\cup}$ denotes the disjoint union and $E^{[s/z]}$ denotes the multiset of E where vertices z are replaced by s, then $G.G'$ can be defined as $G \cdot G' = (V \setminus \{z\} \dot{\cup} V', E^{[s'/z']} \dot{\cup} E', s, z')$. Parallel composition, denoted by $G \parallel G'$ is similar (differing only in that two sources and two sinks are identified) and can be defined as: $G \parallel G' = (V \setminus \{s, z\} \dot{\cup} V', E^{[s'/s, z'/z]} \dot{\cup} E', s', z')$. Figure 2 illustrates the parallel composition of the two graphs given in Figure 1.

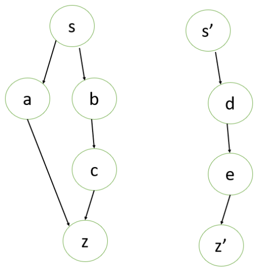

Figure 1. Two source-sink graphs, where a, b, c, d and e are drawn from the set of vertices V, and s, s', z, z' are the unique sources and sinks of the two graphs.

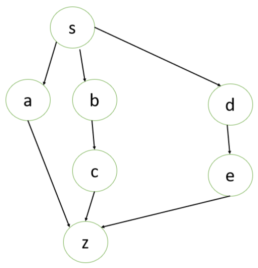

Figure 2. A series parallel graph formed from the parallel composition of the two source sink graphs from Figure 1, with s and s' being merged into s and z and z' being merged into z.

Definition 2. *The set $\mathbb{G}_{\mathcal{SP}}$ over $\mathbb{A}$ is defined inductively by:*

For $a \in \mathbb{A}$, $\xrightarrow{a}$ is an SP graph,
If G and G' are SP graphs, then so are $G \cdot G'$ and $G \parallel G'$.

Hence, the full SP graph semantics for attack tree $\mathbb{T}$ can be given by the function: $[\![\cdot]\!]_{\mathcal{SP}} : \mathbb{T} \to \wp(\mathbb{G}_{\mathcal{SP}})$. This is defined recursively. If $a \in \mathbb{A}, t_i \in \mathbb{T}$, and $1 \leq i \leq k$, then

$[\![a]\!]_{\mathcal{SP}} = \{\xrightarrow{a}\}$
$[\![\mathbf{OR}(t_1, \ldots, t_k)]\!]_{\mathcal{SP}} = \bigcup_{i=1}^{k} [\![t_i]\!]_{\mathcal{SP}}$
$[\![\mathbf{AND}(t_1, \ldots, t_k)]\!]_{\mathcal{SP}} = \{G_1 \,||\, \ldots \,||\, G_k \mid$
$(G_1, \ldots, G_k) \in [\![t_1]\!]_{\mathcal{SP}} \times \ldots \times [\![t_k]\!]_{\mathcal{SP}}\}$
$[\![\mathbf{SAND}(t_1, \ldots, t_k)]\!]_{\mathcal{SP}} = \{G_1 \cdot \ldots \cdot G_k \mid$
$(G_1, \ldots, G_k) \in [\![t_1]\!]_{\mathcal{SP}} \times \ldots \times [\![t_k]\!]_{\mathcal{SP}}\}$ where $[\![t]\!]_{\mathcal{SP}} = \{G_1, \ldots, G_k\}$ corresponds to a set of possible attacks G_i.

As proposed by [50], leaves on an attack tree can be considered as events. The combination of these events can be translated into the processes that form part of a test case. This allows us to use a process algebra such as CSP. The equivalence of the semantics (see Section 3.2) means that we can use synonymous operators to transform a pre-built attack tree.

3.2. CSP

An overview of the subset of CSP that we use is given here. A more complete introduction may be found in [51]. Given a set of events Σ, CSP processes are defined by the following syntax:

$$P \quad ::= \quad Stop \mid e \rightarrow P \mid P_1 \square P_2 \mid P_1;\ P_2 \mid P_1 \underset{A}{\|} P_2$$

where $e \in \Sigma$ and $A \subseteq \Sigma$. For convenience, the set of CSP processes defined via the above syntax is denoted by CSP.

The event $\checkmark \notin \Sigma$ marks successful termination. The process *Stop* is deadlocked. *Skip* is an abbreviation for $\checkmark \rightarrow Stop$. The process $e \rightarrow P$ engages in the event e then behaves as P. The choice $P_1 \square P_2$ behaves either as P_1 or as P_2. The sequential composition $P_1;\ P_2$ initially behaves as P_1 until P_1 terminates, then continues as P_2. The generalised parallel operator $P_1 \underset{A}{\|} P_2$ requires P_1 and P_2 to synchronise on events in $A \cup \{\checkmark\}$. All other events are executed independently, and $P_1 \mid\mid\mid P_2$ is an abbreviation for $P_1 \underset{\emptyset}{\|} P_2$.

There are different semantics models for CSP processes [51]. For our purpose, we recall the finite trace semantics. A *trace* is a possibly empty sequence of events from Σ and may terminate with $\checkmark$. As usual, let Σ^* denote the set of all finite sequences of events from Σ, $\langle\rangle$ the empty sequence, and $tr_1 \frown tr_2$ the concatenation of two traces tr_1 and tr_2; then the set of all traces is defined as $\Sigma^{*\checkmark} = \{tr \frown en \mid tr \in \Sigma^* \wedge en \in \{\langle\rangle, \langle\checkmark\rangle\}\}$.

In general, the trace semantics of a process P is a subset $traces(P)$ of $\Sigma^{*\checkmark}$ consisting of all traces which the process may exhibit. It is formally defined recursively as follows:

- $traces(Stop) = \{\langle\rangle\}$;
- $traces(e \rightarrow P) = \{\langle\rangle\} \cup \{\langle e\rangle \frown tr \mid tr \in traces(P)\}$;
- $traces(P_1 \square P_2) = traces(P_1) \cup traces(P_2)$;
- $traces(P_1;\ P_2) = traces(P_1) \cap \Sigma^*$
  $\cup \{tr_1 \frown tr_2 \mid tr_1 \frown \langle\checkmark\rangle \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$;
- $traces(P_1 \underset{A}{\|} P_2) =$
  $\{tr \in tr_1 \underset{A}{\|} tr_2 \mid tr_1 \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$
  where $tr_1 \underset{A}{\|} tr_2 = tr_2 \underset{A}{\|} tr_1$ is defined as follows with $a, a' \in A \cup \{\checkmark\}$ and $b, b' \notin A$:
  - $\langle\rangle \underset{A}{\|} \langle\rangle = \{\langle\rangle\}$; $\langle\rangle \underset{A}{\|} \langle a\rangle = \emptyset$; $\langle\rangle \underset{A}{\|} \langle b\rangle = \{\langle b\rangle\}$;
  - $\langle a\rangle \frown tr_1 \underset{A}{\|} \langle b\rangle \frown tr_2 = \{\langle b\rangle \frown tr \mid tr \in \langle a\rangle \frown tr_1 \underset{A}{\|} tr_2\}$;
  - $\langle a\rangle \frown tr_1 \underset{A}{\|} \langle a\rangle \frown tr_2 = \{\langle a\rangle \frown tr \mid tr \in tr_1 \underset{A}{\|} tr_2\}$
  - $\langle a\rangle \frown tr_1 \underset{A}{\|} \langle a'\rangle \frown tr_2 = \emptyset$ where $a \neq a'$;
  - $\langle b\rangle \frown tr_1 \underset{A}{\|} \langle b'\rangle \frown tr_2 =$
    $\{\langle b\rangle \frown tr \mid tr \in tr_1 \underset{A}{\|} \langle b'\rangle \frown tr_2\} \cup$
    $\{\langle b'\rangle \frown tr \mid tr \in \langle b\rangle \frown tr_1 \underset{A}{\|} tr_2\}$

As mentioned earlier, trace interleaving is defined as:

- $traces(P_1 \mid\mid\mid P_2) = P_1 \underset{\emptyset}{\|} P_2$

For convenience, we sometimes use the notion of interleaving and concatenation over two sets of traces. In particular, given two sets S_1 and S_2 of traces, $S_1 \mid\mid\mid S_2 = \bigcup_{s_1 \in S_1, s_2 \in S_2} s_1 \mid\mid\mid s_2$ and $S_1 \frown S_2 = \{s_1 \frown s_2 \mid s_1 \in S_1, s_2 \in S_2\}$.

We define $traces^{\checkmark}(P) = \{tr \mid tr \frown \langle\checkmark\rangle \in traces(P)\}$ to denote the set of terminated traces.

A normal way to analyse CSP processes is via trace-refinement. A process P is said to *trace-refine* a process Q (written $Q \sqsubseteq_T P$) if $traces(P) \subseteq traces(Q)$. There are other flavors of refinement, but we restrict ourselves to trace refinement below. In this paper, we use FDR [52] for checking trace-refinements.

In [50], it is showed that each attack tree can be translated into a semantically equivalent CSP process. The equivalence is based on an observation that any series parallel graph (SPG) can be seen as a set of sequences of actions, each corresponding to a traverse from the source node to the sink node of the graph. Formally, the set of sequences of actions of an SP graph can be defined recursively as follows:

- $serialise(\xrightarrow{a}) = \{\langle a \rangle\}$;
- $serialise(G_1 \parallel G_2) = \{s \in s_1 \mid\mid\mid s_2 \mid s_1 \in serialise(G_1) \wedge s_2 \in serialise(G_2)\}$;
- $serialise(G_1 \cdot G_2) = \{s_1 \frown s_2 \mid s_1 \in serialise(G_1) \wedge s_2 \in serialise(G_2)\}$.

The function $serialise(\cdot)$ is also generalised to the case of sets of graphs as follows:

- $serialise(\{G_1, \ldots, G_n\}) = \bigcup_{i \in \{1,\ldots,n\}} serialise(G_i)$.

4. Attack Defense Trees with Sequential Conjunction

Attack defense trees (ADTs) allow us to capture the interaction between attack and defense, and therefore the interaction between attacker and defender. Here we extend ADTs to include sequential conjunction, meaning that we can now capture attacks (or defenses) that have to be carried out in a particular order. This enhanced descriptive power is captured by the operator **SAND**, or sequential **AND**. The resultant tree is called an Attack Defense tree with Sequential conjunction and abbreviated by ADS.

4.1. Syntax

Let $\mathbb{A}$ and $\mathbb{D}$ be disjoint sets of basic attack and defense actions, respectively. Let $\mathbb{B} = \mathbb{A} \cup \mathbb{D}$. The syntax of *attack defense trees with sequential conjunction* (ADS) is given recursively as follows:

$$\begin{array}{lll} t & ::= & t_p \mid t_o \\ t_p & ::= & a \mid t_p \,\mathbf{OR}\, t_p \mid t_p \,\mathbf{AND}\, t_p \mid t_p \,\mathbf{SAND}\, t_p \mid t_p \,\mathbf{C}\, t_o \\ t_o & ::= & d \mid t_o \,\mathbf{OR}\, t_o \mid t_o \,\mathbf{AND}\, t_o \mid t_o \,\mathbf{SAND}\, t_o \mid t_o \,\mathbf{C}\, t_p \end{array}$$

where $a \in \mathbb{A}$ and $d \in \mathbb{D}$. In the above syntax, p stands for proponent, o for opponent, t_p for proponent(attack)-rooted ADS trees and t_o for opponent(defender)-rooted ADS trees. The set of all well-defined attack-rooted ADS trees is denoted by $\mathbb{T}^p_{ADS}$, defense-rooted by $\mathbb{T}^o_{ADS}$, and $\mathbb{T}_{ADS} = \mathbb{T}^p_{ADS} \cup \mathbb{T}^o_{ADS}$. For convenience, we sometimes refer to ADS trees simply as trees unless unclear from the context.

Similar to [53], our main interest is to check if there exists a successful attack (defense) in a given attack(defense)-rooted tree. Here, it is called *ADS tree checking decision problem* and is formally defined as follows:

Definition 3. *Given an ADS tree t, the ADS tree checking problem is to determine if t is successful.*

A more general decision problem is to constrain defense (attack) countermeasures to a subset of the possible ones. This might be valuable if resource constraints meant a limited number of defenses could be employed. However, one can construct a copy t' of t where all defense (attack) countermeasures not in the subset are removed. Then, the general decision problem for t is the same as the decision problem for t'. In the sequel, we develop two formal semantic models for ADS trees and show their equivalence.

4.2. Semantics

The first model is a natural extension of semantics for ADS trees using series parallel (SP) graphs. SP graphs (SPG) have been used to extend the multiset semantics of attack trees to the case of attack trees with the sequential operator. While the extension presented here is natural, it is non-trivial. The later semantics is an elaboration of the former, however, it is closer to the concept of system runs in Computer Science. In particular, each item of the semantics is an interleaving run between attacker and defender.

4.2.1. SPG Semantics

We extend the series parallel graph (SPG) semantics of attack trees with the sequential operator. Particularly, ADS trees are interpreted as sets of SP graphs. Each such set captures possible attacks (resp: defenses) and is associated with a set of SP graphs describing possible defenses (resp: attacks). In the sequel, by abuse of notation, SP graphs shall be denoted in lower case and sets of these in upper case.

Given an attack(defense)-rooted tree t, its semantics is $[\![t]\!]_{SP} = \{(G_1, P_1), \ldots, (G_n, P_n)\}$. Each SP graph $g \in G_i$ canonically represents a set of attacks (respectively: defenses); similarly each SP graph $p \in P_i$ canonically captures a set of defenses (attacks) to counter against any attack (defense) in G_i. For convenience, we upgrade sequential and parallel compositions of SP graphs to sets of SP graphs as follows: $G_1 \circ G_2 = \{g_1 \circ g_2 \mid g_1 \in G_1, g_2 \in G_2\}$ where $\circ \in \{\cdot, ||\}$.

The SPG semantics of ADS trees is given by $[\![\cdot]\!]_{SP} : \mathbb{T}_{ADS} \to \wp(\wp(SP) \times \wp(SP))$ which is defined recursively as follows:

- $[\![b]\!]_{SP} = \{(\{\xrightarrow{b}\}, \emptyset)\}$ for $b \in \mathbb{B}$;
- $[\![t_1 \textbf{ OR } t_2]\!]_{SP} = [\![t_1]\!]_{SP} \cup [\![t_2]\!]_{SP}$;
- $[\![t_2 \textbf{ AND } t_2]\!]_{SP} = \{(G_1 \mid\mid G_2, P_1 \cup P_2) \mid$
  $(G_1, P_1) \in [\![t_1]\!]_{SP}, (G_2, P_2) \in [\![t_2]\!]_{SP}\}$;
- $[\![t_1 \textbf{ SAND } t_2]\!]_{SP} = \{(G_1 \cdot G_2, P_1 \cup P_2) \mid$
  $(G_1, P_1) \in [\![t_1]\!]_{SP}, (G_2, P_2) \in [\![t_2]\!]_{SP}\}$;
- $[\![t_1 \textbf{ C } t_2]\!]_{SP} =$
  $\{(G_1, P_1 \cup (\bigcup_{(G_2,P_2)\in[\![t_2]\!]_{SP}} G_2)) \mid (G_1, P_1) \in [\![t_1]\!]_{SP}\} \cup$
  $\{(G_1 \mid\mid P_2, P_1 \cup (\bigcup_{(G_2',\emptyset)\in[\![t_2]\!]_{SP}} G_2')) \mid$
  $(G_1, P_1) \in [\![t_1]\!]_{SP}, (G_2, P_2) \in [\![t_2]\!]_{SP}, P_2 \neq \emptyset\}$.

Without loss of generality, we provide intuition behind the semantic definition for attack-rooted trees. In the basic case, a single leaf of an attack action b is interpreted as a single pair $(\{\xrightarrow{b}\}, \emptyset)$. The first component means that it is necessary to attack by carrying out b, and the right component means that the defender has no way to counter. The next three cases (**OR, AND** and **SAND**) are self-explanatory, but the last case (**C**) merits attention.

A pair $(G_1, P_1) \in [\![t_1]\!]_{SP}$ means that each attack in t_1 captured by the graph G_1, can be countered by any defense in P_1. Likewise, each defense in t_2, captured by G_2, can be countered by any attack in P_2. Then, in the tree $t_1 \textbf{ C } t_2$, any attack in t_1 can be countered by any defense either in t_1 or in t_2. These attacks therefore are captured in

$$\{(G_1, P_1 \cup (\bigcup_{(G_2,P_2)\in[\![t_2]\!]_{SP}} G_2)) \mid (G_1, P_1) \in [\![t_1]\!]_{SP}\}$$

However, each of such G_2 can be countered again by the corresponding P_2, i.e., $(G_2, P_2) \in [\![t_2]\!]_{SP}$, if $P_2 \neq \emptyset$. Then attacks in G_1 are combined with those from P_2. They can only be countered by defenses without countermeasures in $[\![t_2]\!]_{SP}$, i.e., those in G_2' where $(G_2', \emptyset) \in [\![t_2]\!]_{SP}\}$. They are captured as follows:

$$\{(G_1 \mid\mid P_2, P_1 \cup (\bigcup_{(G_2',\emptyset)\in[\![t_2]\!]_{SP}\setminus\{(G_2,P_2)\}} G_2')) \mid$$
$$(G_1, P_1) \in [\![t_1]\!]_{SP}, (G_2, P_2) \in [\![t_2]\!]_{SP}, P_2 \neq \emptyset\}$$

As a simple example of the last case, consider the ADS tree in Figure 3. This contains the **SAND** operator and given by the SPG semantics $\{(\{\xrightarrow{a_1} \cdot \xrightarrow{a_2}\}, \{\xrightarrow{d_1}, \xrightarrow{d_2}\})\}$. In this case the attack a_1 can be countered by defense d_1, and the attack a_2 by defense d_2. The **SAND** operator means that *both* attacks a_1 and a_2 need to succeed in order for the overall attack A to be successful.

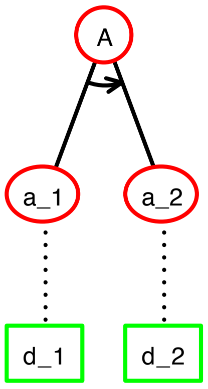

Figure 3. An Attack Defense Tree containing the **SAND** operator, and given by the SPG semantics $\{(\{\xrightarrow{a_1} \cdot \xrightarrow{a_2}\}, \{\xrightarrow{d_1}, \xrightarrow{d_2}\})\}$. In this case the attack a_1 can be countered by defense d_1, and the attack a_2 by defense d_2.

Now we build a representation of a defense mechanism, depicted in Figure 4, that we wish to add to the attack tree from Figure 5. This defense mechanism is made up of two components: d_3 and d_4, and is a defense against the whole attack (represented by A.) Within it, there is a counterattack against d_3 (a_3) but not against d_4. The SPG semantic presentation of the whole attack defense tree uses the operator **C** and is given by $(A \mathbf{C} D)$. The SPG semantics of this graph is given by $\{(\{\xrightarrow{d_3}\}, \{\xrightarrow{a_3}\}), (\{\xrightarrow{d_4}\}, \emptyset)\}$. We see the graphical result when the defense is combined with the attack in Figure 5.

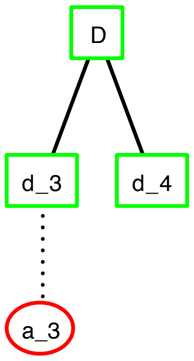

Figure 4. The constructed defense is made up of two components: d_3 and d_4. A counterattack against d_3 exists (attack a_3) but no counterattack against d_4 exists.

The additional defense mechanism D is capable of defending against the combined attack a_1 followed by a_2. The SPG semantics of the overall tree (given by A **C** D) expands to the set of two pairs $\{(\{\xrightarrow{a_1} \cdot \xrightarrow{a_2}\}, \{\xrightarrow{d_1}, \xrightarrow{d_2}, \xrightarrow{d_3}, \xrightarrow{d_4}\}), (\{(\xrightarrow{a_1} \cdot \xrightarrow{a_2}) \,||\, \xrightarrow{a_3}\}, \{\xrightarrow{d_1}, \xrightarrow{d_2}, \xrightarrow{d_4}\})\}$.

In this example, we can see that any of defense actions $\xrightarrow{d_1}$, $\xrightarrow{d_2}$ or $\xrightarrow{d_4}$ is sufficient to defend against either of the two attacking options ((a_1 AND a_2) OR a_3) open to the attacker.

Given the semantics of ADS trees, we now formalise the analysis question of interest. An attack (defense) on a given ADS tree is successful if and only if there exists a pair $(G, \emptyset) \in [\![t]\!]_{SP}$, i.e., attacks (defenses) captured in G have no corresponding countermeasures.

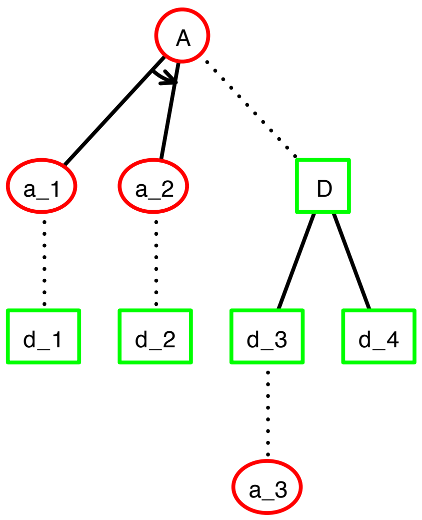

Figure 5. The Defense combined with the Attack Tree. The defense D is a defense against the total attack (A). The SPG semantic presentation of this Attack Defense Tree is $(A \ \mathbf{C} \ D)$.

4.2.2. Trace Semantics

Any SP graph can be seen as a set of sequences of actions by traversing from the source node to the target. Therefore, we extend the function $serials(\cdot)$, presented in [50], to the function $serialise(\cdot)$, as below.

First, recall that the purpose of the function $serials(\cdot)$ is to construct a serial representation of the actions from a SP graph. Parallel graph composition is considered as interleaving and sequential composition as concatenation.

The purpose of the function $serialise(\cdot)$ is to construct all these combinations of actions as follows.

Definition 4. *Given G in $\mathbb{G}_{\mathcal{SP}}$, let serials$(G)$ denote the set of all possible ways to serialise G as follows:*

- $serialise(G)) = \{s \in serials(g) \mid g \in G\}$;
- $serialise(G, P) = (serialise(G), serialise(P))$; *and*
- $serialise(\{(G_1, P_1), \ldots, (G_n, P_n)\}) = \bigcup_{i \in \{1,\ldots,n\}} \{serialise(G_i, P_i)\}$.

Therefore, another way to interpret trees is to serialise the SP graph semantics, i.e., to interpret them as sets of pairs of the form $(S, T) \in \wp(\mathbb{A}^+) \times \wp(\mathbb{D}^*) \cup \wp(\mathbb{D}^+) \times \wp(\mathbb{A}^*)$ where S represents a set of attacks (defenses) if t is attack(defense)-rooted and T the corresponding countermeasures. To this end, we are effectively defining a trace semantics for ADS trees.

The trace semantics of ADS trees is given by the function $[\![\cdot]\!]_T : \mathbb{T}_{ADS} \to \wp(\wp(\mathbb{A}^+) \times \wp(\mathbb{D}^*) \cup \wp(\mathbb{D}^+) \times \wp(\mathbb{A}^*))$ which is defined recursively as follows:

- $[\![b]\!]_T = \{(\{\langle b\rangle\}, \emptyset)\}$ for $b \in \mathbb{B}$;
- $[\![t_1 \ \mathbf{OR} \ t_2]\!]_T = [\![t_1]\!]_T \cup [\![t_2]\!]_T$;
- $[\![t_1 \ \mathbf{AND} \ t_2]\!]_T = \{(S_1 \ ||| \ S_2, T_1 \cup T_2) \mid$
  $(S_1, S_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T\}$;
- $[\![t_1 \ \mathbf{SAND} \ t_2]\!]_T = \{(S_1 \frown S_2, T_1 \cup T_2) \mid$
  $(S_1, T_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T\}$;
- $[\![t_1 \ \mathbf{C} \ t_2]\!]_T =$
  $\{(S_1, T_1 \cup (\bigcup_{(S_2,T_2) \in [\![t_2]\!]_T} S_2)) \mid (S_1, T_1) \in [\![t_1]\!]_T\} \cup$
  $\{(S_1 \ ||| \ T_2, T_1 \cup (\bigcup_{(S_2',\emptyset) \in [\![t_2]\!]_T} S_2')) \mid (S_1, T_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T, T_2 \neq \emptyset\}$.

Unsurprisingly, this semantics is equivalent to the SPG semantics once we serialise all sequences in a GP graph. We then have the following theorem.

Theorem 1. $\forall \, t \in \mathbb{T}_{ADS} : serialise([\![t]\!]_{SP}) = [\![t]\!]_T$.

The following result is immediate.

Corollary 1. $\forall\, t \in \mathbb{T}_{ADS} : \exists (G, \varnothing) \in [\![t]\!]_{SP} \Leftrightarrow (S, \varnothing) \in [\![t]\!]_{T}$.

In summary, this means that to check if an attack depicted by a tree t is successful, it is sufficient to check the existence of $(S, \varnothing) \in [\![t]\!]_{T}$.

5. Reasoning about ADS Trees

An algorithm is presented for the ADS tree checking problem. To know which are successful attacks (or defenses) for a given ADS tree, a translation from ADS trees to CSP processes is introduced. Then, FDR [52], CSP's model checker, is employed to elicit a successful attack (defense).

5.1. Checking ADS trees

Given the previous result, we propose a simple algorithm, Algorithm 1, for the ADS tree checking problem. It is similar to the Boolean semantics of ADS trees [54] where the difference between **AND** and **SAND** is discarded. The reason is that the checking problem is concerned with the existence of a successful attack (defense) rather than the details of it's construction. Note that b is an leaf node in the tree to be checked.

Algorithm 1: Checking ADS trees.

```
Function $check(t)$
    input  : An ADS tree $t$;
    output : $true$ if $t$ is successful, $false$ otherwise.
    if $t = b$ then
        return $true$;
    else if $t = t_1$ OR $t_2$ then
        return $check(t_1) \vee check(t_2)$;
    else if $t = t_1$ AND $t_2$ or $t = t_1$ SAND $t_2$ then
        return $check(t_1) \wedge check(t_2)$;
    else if $t = t_1$ C $t_2$ then
        return $check(t_1) \wedge \neg check(t_2)$;
    end
end
```

The following correctness result is immediate.

Lemma 1. *Given* $t \in \mathbb{T}_{ADS}$, $check(t) = true$ *iff* $\exists (S, \varnothing) \in [\![t]\!]_{T}$.

Since Algorithm 1 visits all the nodes of an input ADS tree t, its complexity is $O(n)$ where n is the number of nodes of t.

5.2. Translation to CSP

Each ADS tree t is translated into a set of CSP process pairs where the alphabet $\Sigma_t = \mathbb{B}$. The translation function $trans(\cdot)$ is defined as follows:

- $trans(b) = \{(b \rightarrow Skip, Stop, \top)\}$
- $trans(t_1\ \mathbf{OR}\ t_2) = trans(t_1) \cup trans(t_2)$
- $trans(t_1\ \mathbf{AND}\ t_2) = \{(P_1 \mid\mid\mid P_2, Q_1 \Box Q_2, O_1 \wedge O_2) \mid$
  $(P_1, Q_1, O_1) \in trans(t_1), (P_2, Q_2, O_2) \in trans(t_2)\}$

- $trans(t_1\,\mathbf{SAND}\,t_2)\{(P_1;\ P_2, Q_1 \sqcap Q_2, O_1 \wedge O_2) \mid$
  $(P_1, Q_1, O_1) \in trans(t_1), (P_2, Q_2, O_2) \in trans(t_2)\}$

- $trans(t_1\,\mathbf{C}\,t_2) = \{(P_1, Q_1 \sqcap (\Box_{(P_2,Q_2,O_2)\in trans(t_2)} P_2), \bot) \mid (P_1, Q_1, O_1) \in trans(t_1)\}$
  $\cup\{(P_1 \mid\mid\mid Q_2, Q_1 \sqcap (\Box_{(P_3,Q_3,\top)\in trans(t_2)} P_3), O) \mid$
  $(P_1, Q_1, O_1) \in trans(t_1), (P_2, Q_2, O_2) \in trans(t_2),$
  $O = O_1 \wedge \neg\exists(P_3, Q_3, \top) \in trans(t_2)\}$

The translation is similar to the definition of the trace semantics for ADS tree. Intuitively, $trans(t)$ aims to translate ADS trees t into triples of two CSP processes and an outcome. The outcome is either success/true, denoted by $\top$, or failure/false, denoted by $\bot$. The following argument is applied for an attack-rooted tree t. If t is defense-rooted, the same argument is also applied where "attack" is exchanged with "defense".

Given such a triple $(P, Q, O) \in trans(t)$, each terminated trace $tr_1 \in traces^{\checkmark}(P_1)$ is an attack while each terminated trace $tr_2 \in traces^{\checkmark}(Q)$ is a defense against tr_1. When $traces^{\checkmark}(Q) = \emptyset$, every attack in $traces^{\checkmark}(P)$ succeeds, hence $O = \top$. When $traces^{\checkmark}(Q) \neq \emptyset$, every trace in $traces^{\checkmark}(P)$ can be countered by any trace in $traces^{\checkmark}(Q)$, hence $O = \bot$.

In the basic case, $t = b$ where $b \in \mathbb{A}$, performing b will ensure success where no countermeasure is available. For $t = t_1\,\mathbf{OR}\,t_2$, all triples in $trans(t_1)$ and $trans(t_2)$ are collected by union. For $t = t_1\,\mathbf{AND}\,t_2$, an attack can be constructed by an interleaving of an attack in P_1 of $trans(t_1)$ with another in P_2 of $trans(t_2)$. They can be countered by a defense in either Q_1 of $trans(t_1)$ or Q_2 of $trans(t_2)$.

These combinations are captured by $(P_1 \mid\mid\mid P_2, Q_1 \sqcap Q_2, O_1 \wedge O_2)$ where $(P_i, Q_i, O_i) \in trans(t_i)$ for $i \in \{1, 2\}$. The same explanation applies for $t = t_1\,\mathbf{SAND}\,t_2$ by replacing $\mid\mid\mid$ with "; " when combining attacks in $trans(t_1)$ and $trans(t_2)$.

Finally, for $t = t_1\ \mathbf{C}\ t_2$, attacks in P_1 of $trans(t_1)$ can be countered by a defense in Q_1 of $trans(t_1)$ itself or defenses in P_1 of $trans(t_2)$. These attacks are captured by $(P_1, Q_1 \sqcap (\Box_{(P_2,Q_2,O_2)\in trans(t_2)} P_2), \bot)$ where (P_1, Q_1, O_1) of $trans(t_1)$. However, these defenses in P_2 of $trans(2)$ can be countered by an attack in Q_1 of $trans(t_2)$ if available, i.e., if $traces^{\checkmark}(Q_2) \neq \emptyset$. This attack can be combined with that in P_1. The combination can only be countered (i) by defenses in Q_1 or (ii) by defenses in P_3 of $trans(t_2)$) which are not countered by any attack, i.e., $(P_3, Q_3, \top) \in trans(t_2)$. These combined attacks are captured by $(P_1 \mid\mid\mid Q_2, Q_1 \sqcap (\Box_{(P_3,Q_3,\top)\in trans(t_2)}), O)$ where $(P_1, Q_1, O_1) \in trans(t_1)$, $(P_2, Q_2, \bot) \in trans(t_2)$ and $O = O_1 \wedge \neg\exists(P_3, Q_3, \top) \in trans(t_2)$.

We define $traces(trans(t)) = \{(\{tr_1 \mid tr_1 {}^\frown \langle\checkmark\rangle \in traces(P)\}, \{tr_2 \mid tr_2 {}^\frown \langle\checkmark\rangle \in traces(Q)\} \mid (P, Q, O) \in trans(t)\}$. The following result is immediate.

Lemma 2. *$\forall\, t \in \mathbb{T}_{ADS} : (P, Q, \top) \in trans(t) \Leftrightarrow traces^{\checkmark}(Q) = \emptyset$.*

Then, the correctness of the translation from ADS trees to CSP processes is stated below.

Theorem 2. *$\forall\, t \in \mathbb{T}_{ADS} : traces(trans(t)) = [\![t]\!]_T$.*

5.3. Automated Reasoning via CSP Refinements

In this section, we show how to use refinement on the translation of ADS trees into CSP processes to reason about ADS trees. Our main interest is to answer the question whether an attack(defense)-rooted tree is successful. By Lemma 2, the following is immediate.

Corollary 2. *$\forall\, t \in \mathbb{T}_{ADS} : \exists(S, \emptyset) \in [\![t]\!]_T \Leftrightarrow \exists(P, Q, \top) \in trans(t)$.*

This result means that to determine if an ADS is successful, we need to find a triple $(P, Q, \top) \in trans(t)$. The set of all terminated traces without countermeasures in t then is captured by $\Box_{(P,Q,\top)\in trans(t)} P$. The set of non-terminated traces is captured by $Run(\mathbb{A})$ if t

is attack-rooted or $Run(\mathbb{D})$ if otherwise. Then, terminated traces without countermeasures can be identified by the following refinement:

- $Run(\mathbb{A}) \sqsubseteq_T trans(t) \;\Box_{(P,Q,\top)\in trans(t)} P$ if $t \in \mathbb{T}^p_{ADS}$;
- $Run(\mathbb{D}) \sqsubseteq_T trans(t) \;\Box_{(P,Q,\top)\in trans(t)} P$ if $t \in \mathbb{T}^o_{ADS}$.

If the refinement is true, $\Box_{(P,Q,\top)\in trans(t)} P$ has no terminated traces, i.e., $trans(t)$ has no triple in $(P, Q, \top)$. Therefore, t is not successful. Conversely, if the refinement fails, i.e., there exists a counter example and so t is successful. By using FDR, this refinement can be automatically checked. If it fails, a counter example will be produced which is evidence of an attack (or defense if t is defense-rooted) without countermeasures.

6. Generating ADS Trees with SAND

6.1. Methodology

We propose a method to generate ADS trees from a system's network model and a library of attack defense trees. Each tree in the library is used to capture weaknesses and/or vulnerabilities which lead to specific attacks on concrete components or sub-systems. In this paper, they are called templates. The overall ADS tree generation process is described in Figure 6. It comprises four stages:

- Stage 1—Construction of frame attack tree - a tree, which shows attack paths, generated from the system's network model.
- Stage 2—ADS tree templates are selected from the library and attached to the matching nodes of frame tree.
- Stage 3—ADS tree is revised and any missing attacks or defenses are added.
- Stage 4 (optional)—The library is updated to include newly identified templates.

The system's network model is used as an input to stage 1. It should include system components and their communications links, to be considered in ADS tree. For example, the network model of an automated vehicle comprises the following elements:

- Hardware devices (HW): Electronic Control Units (ECU), on-board computers, sensors, actuators, network devices (switch, router), etc.
- Communication networks (CN): wired and wireless networks, such as Ethernet, Wi-Fi, Controlled Area Network (CAN) bus, etc.
- Connections (CO): point-to-point connections between devices.
- Access points to devices and communication networks (AP): USB, Bluetooth, OBD, etc.

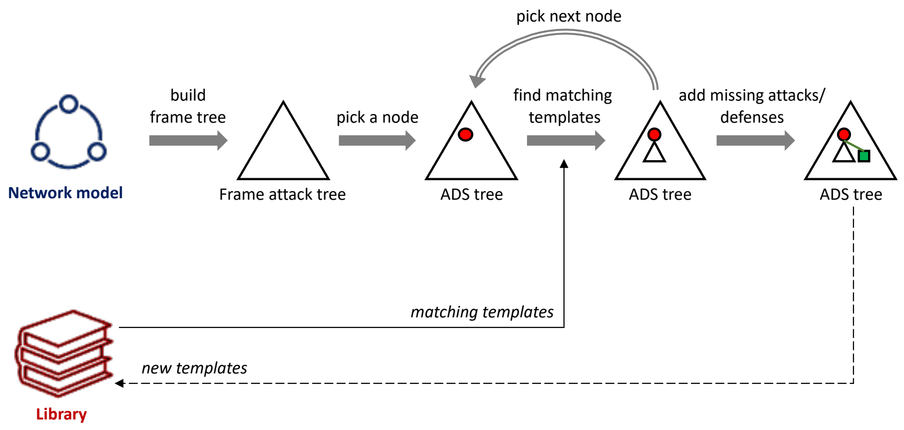

Figure 6. Methodology. Construction of frame attack tree is generated from system's network model; ADS tree templates are selected from library and attached to matching nodes of frame tree; ADS tree is revised and any missing attacks or defenses are recursively added; Library is optionally updated to include newly identified templates.

To construct frame attack tree, we select the network model element, whose vulnerabilities we want to analyse (Element X) and place it at the root of the tree. We name the root node "Compromise Element X". Then, we traverse to neighbouring elements (HW, CN, CO, and AP) and add them to the frame as the child nodes. The process continues until all reachable network model's elements have been added to the frame tree.

In stage 2, the frame attack tree, constructed during stage 1, is expanded by adding matching ADS tree templates from the library. Two types of templates could be used: attack templates and defense templates. Attack templates expand the node into child attack and defense nodes. Child attack nodes provide more details on how the security properties of tree nodes – Confidentiality, Integrity, and Availability (C/I/A)—could be violated, for example, the node "Compromise Element Y" could be matched with the template "Violate C/I/A of Element Y".

Defense template includes an attack node and corresponding defense nodes. Defenses nodes show how an attack node can be mitigated. They could include additional information, such as protection type (prevention, reduction, detection, correction, or monitoring of losses or compromise, and restoring or recovery from damage) [2] and protected security properties. The process of expanding ADS tree continues until all frame tree's nodes have been analysed and matching templates from the library have been assigned.

The aim of stage 3 is to review the ADS tree, generated in stage 2 and identify any inconsistencies and/or missing information. For example, we can use the reasoning algorithm introduced in Section 5 to check for undefended attacks. This can be done any time the external environment changes and new attacks are discovered.

At this stage, additional attack and defense nodes could be manually added to ADS tree.

Stage 4 is optional, aimed at updating the library to include new templates from the ADS tree, constructed through stages 1–3.

This methodology could be used during the concept phase of automotive cybersecurity lifecycle, defined by the ISO/SAE 21434 standard, for:

- determining risk level of identified threat scenarios during Threat Analysis and Risk Assessment (TARA) process. Risk value is calculated based on attack impact and feasibility ratings. One of the approaches for determining threat feasibility—"attack vector-based approach"—uses attack path analysis for estimating threat feasibility. ADS trees provide detailed information on attack paths and available defenses. In general, defenses are not considered during TARA. However, if the information on defenses is available in ADS trees, it could be used for more precise feasibility evaluation, which takes into consideration both attacks and defenses.
- verifying that the set of defenses, selected for implementation, is sufficient for protecting the system, and that the residual risk has been reduced to an acceptable level. At the end of concept phase, when safety measures have been assigned to threats, ADS tree could be updated to include security measures, selected for implementation to analyse attack coverage by defense. Then, threat feasibility ratings could be revised and residual risk determined by combining revised feasibility and impact ratings.

Each developer (e.g., Original Equipment Manufacturer (OEM) in automotive industry) should have its own library of attack and defense templates and reuse it across different projects and products while building their ADS trees. The library could be useful for continuous vulnerability monitoring as well. For example, if a new template is defined for a certain attack node, all ADS trees that use that attack node should be updated.

6.2. ADS Tree Generation Example

A hypothetical autonomous vehicle example is used to demonstrate the ADS tree generation. Its network model is shown in Figure 7. The main elements include on-board computer, which uses sensor (camera, Lidar, GPS, Radar) signals as an input to compute vehicle control signals. The control signals are then sent via CAN bus to ECU, which transforms them into control commands and sends them for implementation to actuators (steering, braking, and acceleration). This example includes several network types (CAN bus, Ethernet), various access points (USB, Bluetooth, OBD), and a physical connection between ECU and actuators.

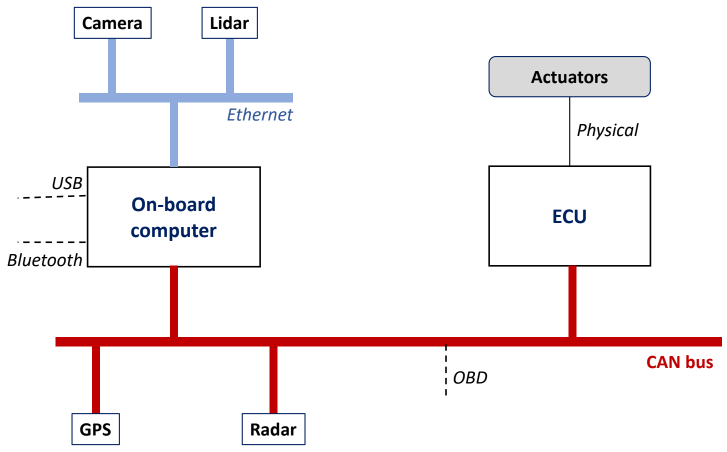

Figure 7. A hypothetical autonomous vehicle network model which will be used to demonstrate ADS tree generation.

We build the frame attack tree manually, starting with "Compromise on-board computer" as a root node (see Figure 8). Then, we add the child nodes corresponding to attacks on the elements, connected to on-board computer, followed by attacks on the elements connected to these elements and so on, until all reachable network model's elements have been included. Note that only cyberattacks are considered in Figure 8. Thus, network model's elements, which are not susceptible to cyberattacks (physical connections and actuators) are not included in the frame tree.

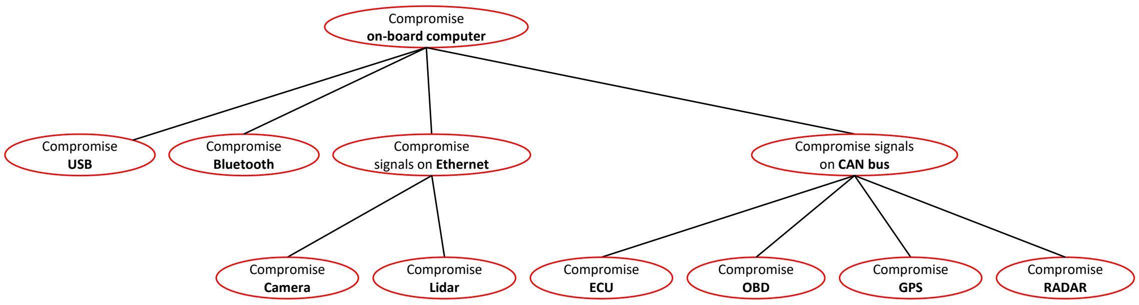

Figure 8. A frame attack tree beginning with attacks on the Onboard computer. System elements not susceptible to cyber attacks are not included here.

Then, we search the library for ADS tree templates and add them to frame attack tree. Five matching templates have been identified, as shown in Figure 9: two attack templates (violate integrity on USB, and violate confidentiality and integrity on OBD) and three defense templates (Ethernet defenses, On-board computer defenses, and CAN bus defenses). Attack templates are shown in Figure 10, while defense templates are depicted in Figure 11.

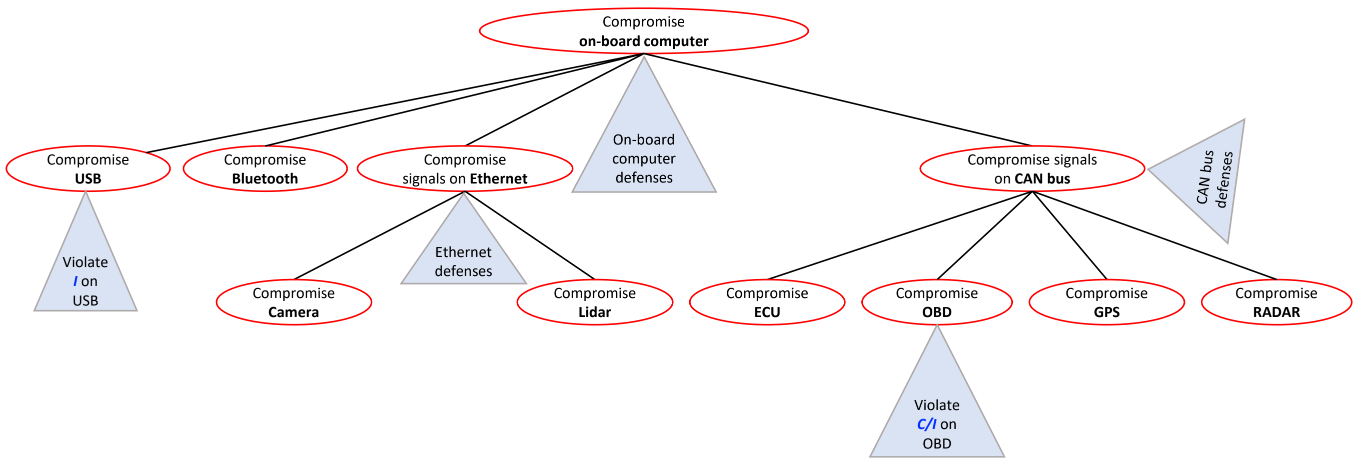

Figure 9. Extending the frame attack tree with templated attacks (represented with triangles).

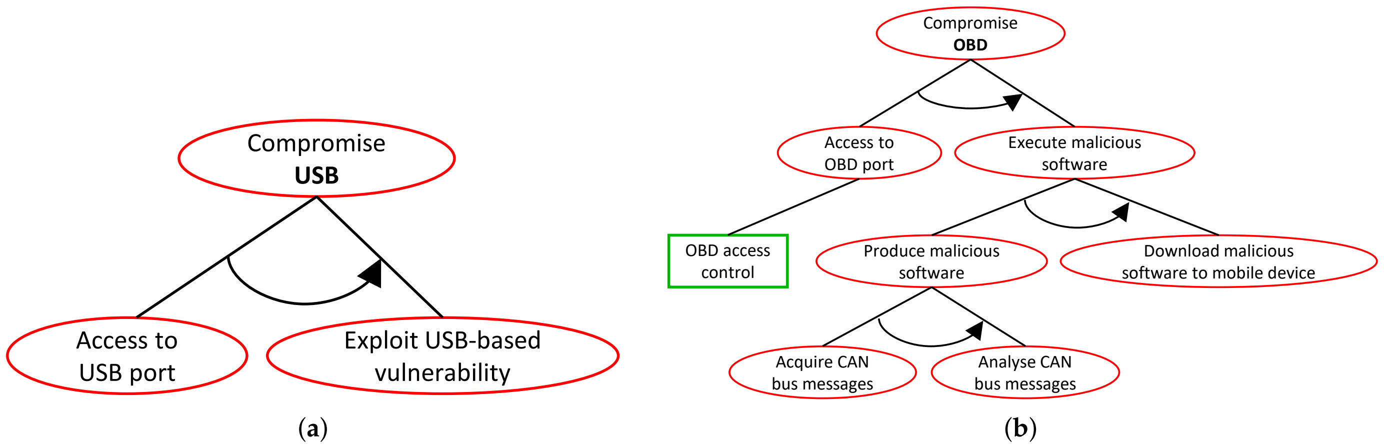

Figure 10. ADS tree attack templates. (**a**) An attack resulting in an integrity violation on USB. (**b**) An attack resulting in a confidentiality and integrity violation on USB.

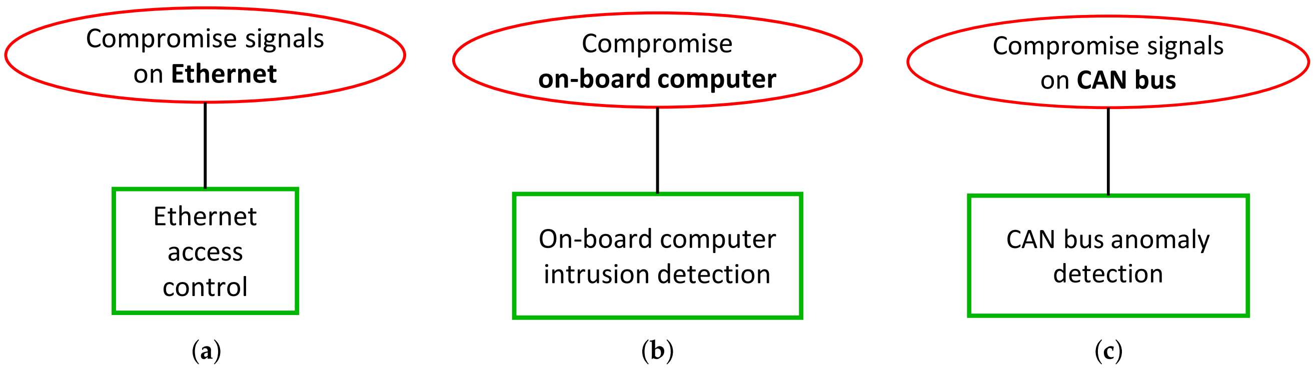
Compromise signals on Ethernet
Ethernet access control
(a)
Compromise on-board computer
On-board computer intrusion detection
(b)
Compromise signals on CAN bus
CAN bus anomaly detection
(c)

Figure 11. A selection of ADS tree defence templates. (**a**) Ethernet defences. (**b**) Onboard computer defences. (**c**) CAN bus defences.

6.3. Implementation and Experiment

In this section, we present the algorithm to automate the first two stages of the generation method proposed in Section 6.1. This algorithm takes three inputs: a structural model of a system, a library of ADS tree templates and a target component within the system, and constructs an ADS tree where the root is to compromise to the provided target component. The pseudo-code of the algorithm is presented in Algorithm 2.

Algorithm 2: Generating attack defense trees.

```
Function GenerateADST(Model, Library, Target)
    tree ← BuildFrameTree(Model, Target);
    queue ← GetNodes(tree);
    while queue is not empty do
        node ← pop(queue);
        subTrees ← getMatchingTemplates(node, Liberary);
        subAttackTrees ← filterAttackTrees(subTrees);
        if subAttackTrees is not empty then
            if node is not leaf and of type AND or SAND then
                copied_subtree ← copy of the subtree at node;
                change type of node to be OR;
                set branches of node to be the copied_subtree and all trees in
                  subAttackTrees;
            else
                add all trees in subAttackTrees to be branches of node;
            end
        end
        for each subtree ∈ subtrees that is not in subAttackTrees do
            add all subtrees of subtree as branches of node;
        end
        add all newly added nodes below node to queue;
    end
    return tree;
end
```

The algorithm starts at line 2 with building a frame attack tree, as described by stage 1 of the methodology. This is based on the input model and the target component. Building such a frame attack tree is carried out in a breadth-first search fashion through the model. From the target component, the search keeps visiting networks and nodes that are reachable. The implementation of this process is depicted in Algorithm 3.

Algorithm 3: Building the frame tree.

```
Function BuildFrameTree(Model, Target)
    root ← Node with label Compromise plus Target;
    queue ← all pairs of (root,network) for each network that Target connects to;
    mark root as visited;
    while queue is not empty do
        (tree,net_or_comp) ← remove the first element of queue;
        add a new node with label Compromise net_or_comp as a child of tree;
        mark net_or_comp as visited;
        if net_or_comp is a network then
            add all pairs (new node, component) for each non-visited component
              connected to net_or_comp;
        else
            add all pairs (new node, network) for each non-visited network that
              net_or_comp and connects to;
        end
    end
    return root;
end
```

After building the frame tree, Algorithm 2 realises stage 2 by browsing through each node of the tree to find out if any template from the library can be used to expand it. In particular, line 3 simply adds all nodes of the frame tree to the queue and consider each node in the queue one by one. For each node in the queue, all matching templates in the library are collected at line 6. A template matches a node if its root's label matches the node's label. These templates are separated into those with at least one attack node immediately below their roots, called attack templates, and those without. The attack templates are added as branches of the currently considered node as an alternative way to attack. This corresponds to lines 8 to 16 of Algorithm 2.

The other templates are added as countermeasures branches to the nodes, as presented in lines 17 to 19.

It is straightforward that Algorithm 3 terminates due to the finiteness of the input model. Since every element of the input model is marked as visited whenever considered at line 8, and they will never be considered twice. Eventually, all reachable elements from the input target must be visited. Then, the queue will be empty and the condition on line 5 will return false. In this case, the algorithm will return the frame tree and terminate.

The termination of Algorithm 2 is more involved. To ensure termination, the library of ADS tree templates is required to be finite and not to contain a looped sequence of templates. A template is said to be the successor of another if its root matches a node of the other template. When this matching node is considered at line 5 of Algorithm 2, the successor template will be selected to extend the ADS tree by line 6. A sequence of templates consists of a list of templates such that the later is the successor the former in the list. A sequence of templates contains a loop if a template occurs more than once in the sequence. By requiring the input library to be finite and to contain no such loop, the termination of Algorithm 2 can be proved by contradiction. In fact, let us assume that the algorithm does not terminate, this means it will generate an infinite ADS tree. In other words, there exists an infinite branch along this tree which is obtained by using an infinite sequence of templates. Since the library is finite, there must be a template that occurs infinitely many times on this sequence, i.e., this sequence contains a loop. This contradicts the requirement for the library.

Experiment and Evaluation

This implementation (accessible at https://tinyurl.com/genadstree, accessed on 27 July 2023) is used to generate ADS trees for the example presented in Section 6.2. For each component in the example, we run Algorithm 2 with the input model as depicted in Figure 7 and the library consisting of templates as provided in Figures 10 and 11. For each execution, the algorithm produces an output ADS tree. These trees contain 25 nodes with a height ranging between 7 and 9. Figure 12 shows the obtained ADS tree when selecting OBCOMPUTER (i.e., onboard computer) as the target. The algorithm generates the ADS tree as expected. The root node, labelled "Compromise OBCOMPUTER", is expanded according to BuildFrameTree (Algorithm 3) with all networks/interfaces connected to the OBCOMPUTER. This network includes BLUETOOTH, USB, ETHERNET and CAN according to the vehicle network model (see Figure 8). Once they are compromised, the next targets by an attacker will be the components attached to these networks (excluding the OBCOMPUTER). In particular, for ETHERNET, the next targets are Camera and Lidar; for CAN, the next targets are OBD, RADAR, GPS and the ECM (Engine Control Module) ECU. As no further network beyond these components is available according to the network model, BuildFrameTree terminates, and the generation algorithm continues building the attack tree based on available templates. In particular, the template for compromising USB (Figure 11a) is instantiated and attached to the node "Compromise USB" of the frame tree; the template for compromising OBD (Figure 11b) is instantiated and attached to the node "Compromised OBD". Furthermore, the defense templates from the input library are used to expand the attach tree with defense nodes. In particular, the template for defending against compromising ETHERNET (Figure 12) is attached to the node "Compromise ETHERNET";

the template for defending against compromising OBDCOMPUTER (Figure 12) is attached to the node "Compromise OBDCOMPUTER"; and the template for defending against compromising CAN (Figure 12) is attached to the node "Compromise CAN". Overall, the generated attack tree (Figure 12) captures all possible attacks on the input vehicle models and makes use of all possible templates from the input library.

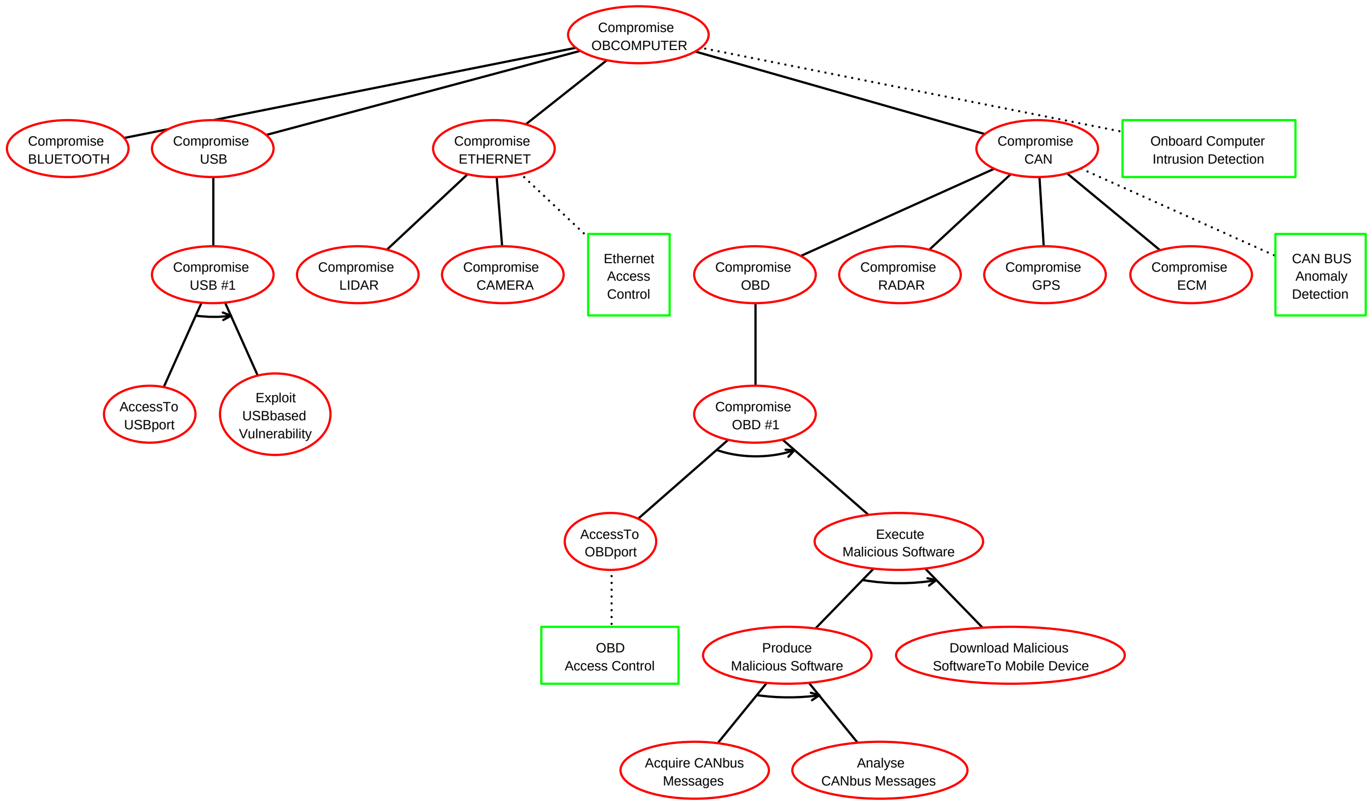

Figure 12. The final generated ADS tree targeting the Onboard Computer.

7. Conclusion and Further Work

In this paper we have given a formal ADS (Attack Defense with Sequential AND) tree representation, a reasoning algorithm to allow checking whether attacks are defended against, and a methodology for automated generation of ADS trees. The methodology has been demonstrated using a simple hypothetical example inspired from the architecture of an autonomous vehicle. The example methodology includes as well simple templates that have been constructed to demonstrate simple attacks and defenses.

ADS trees are useful in both CPS development and post-development phases. In the development phase, they aid in performing cybersecurity risk analysis, while in post-development phases, such as operation and maintenance, they can be used for monitoring and maintaining the required security level. For example, it is known that vehicle manufacturers will have to support new vehicles (including updating the security case) for many years after the vehicle has been being produced. During this time new attacks and vulnerabilities will (likely) be discovered. This could potentially mean that the security case will have to become a "living document". See for example the work by Cobos et al. [55] which describes how ADS could be combined with Goal Structuring Notation for building the automotive cybersecurity assurance case. The methodology we present could help to automate part of this process.

Using our template-based methodology will mean that as new attacks become identified they can be described as templates and added to the template library, together with any relevant defenses that have been identified to date. The new ADS can then be used to re-build and update the new security case for the updated vehicle.

All nodes, except those denoting defenses, of our templates (or even the generated ADS trees) describe the actions/goals from the attacker's perspective. This gives a clean diagram for the analyst to focus on and comprehend the attacker's course of actions. However, it may fail to capture other crucial events (such as hardware component failures that are indirectly or not necessarily caused by attacker) that can cause vulnerabilities which the attacker can take advantage of. Such attack-contributing/triggering events are in fact possible occurrences especially in CPS. Consider this simple example; a malware running stealthily on the in-vehicle infotainment system would not perform anything harmful until the (legitimate) user performs particular actions (e.g., pressing a button or turning the radio node); note that such actions are not performed by attacker but they unintentionally trigger or contribute to the attack. Therefore, such hazardous, accidental or attack-contributing/triggering events (caused by parties other than attacker and defender) should also be explicitly captured by the ADS trees or templates, making sure that they are not missed by the analysts when devising the mitigation measures/strategies. The resultant ADS trees or templates may be adapted from the Attack-Fault Trees [56,57] or Exploit Trees [41].

A potential further step could also be optimization of size or structure of the generated ADS trees via mechanisms adapted from or inspired by, for instance [34]'s logic reduction techniques, and refinement rules of [17,19,35]. Additionally, automated quantitative analysis may also be explored considering a plethora of techniques available in the literature (cf. Table 4 of [21]). To this end, we will investigate how ADS can help to analyse common cyber security metrics including attack success probabilities and cost of successful attacks. Another step is to establish libraries of templates so that the proposed method can be applied. Existing attack pattern and vulnerability databases (such as CAPEC [40], CWE [58], NVD [59], etc.) will be good starting points to build such libraries.

Author Contributions: Conceptualization, J.B., H.N.N. and G.S.; Methodology, J.B., G.S. and S.A.S.; Validation, J.B., H.N.N., G.S. and S.A.S.; Formal analysis, J.B. and H.N.N.; Writing—original draft, J.B., H.N.N., G.S. and S.A.S.; Writing—review & editing, J.B., L.S.L. and G.S.; Supervision, S.A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Schneier, B. AT: Modeling Security Threats. Available online: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (accessed on 1 July 2023).
2. *BS ISO/SAE 21434:2021*; Road Vehicles—Cybersecurity Engineering. International Organization of Standardization: Geneva, Switzerland, 2021.
3. Kordy, B.; Piètre-Cambacédès, L.; Schweitzer, P. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Comput. Sci. Rev.* **2014**, *13–14*, 1–38.
4. Kordy, B.; Wideł, W. On Quantitative Analysis of Attack–Defense Trees with Repeated Labels. In *Principles of Security and Trust*; Bauer, L., Küsters, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10804, pp. 325–346. [CrossRef]
5. Arnold, F.; Hermanns, H.; Pulungan, R.; Stoelinga, M. Time-dependent analysis of attacks. In Proceedings of the Third International Conference on Principles and Security of Trust, POST 2014, Grenoble, France, 5–13 April 2014; pp. 285–305. [CrossRef]
6. Jhawar, R.; Kordy, B.; Mauw, S.; Radomirović, S.; Trujillo-Rasua, R. Attack Trees with Sequential Conjunction. In *ICT Systems Security and Privacy Protection*; Springer: Berlin/Heidelberg, Germany, 2015; Volume 455, pp. 339–353. [CrossRef]
7. Ivanova, M.G.; Probst, C.W.; Hansen, R.; Kammüller, F. Transforming graphical system models to graphical attack models. In Proceedings of the Second International Workshop, GraMSec 2015, Verona, Italy, 13 July 2015; pp. 82–96.
8. Jürgenson, A.; Willemson, J. Serial Model for Attack Tree Computations. In Proceedings of the 12th International Conference, Seoul, Republic of Korea, 2–4 December 2009; Lee, D., Hong, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 118–128.
9. Piètre-Cambacédès, L.; Bouissou, M. Beyond Attack Trees: Dynamic Security Modeling with Boolean Logic Driven Markov Processes (BDMP). In Proceedings of the 2010 European Dependable Computing Conference, Valencia, Spain, 28–30 April 2010; pp. 199–208. [CrossRef]

10. Bistarelli, S.; Fioravanti, F.; Peretti, P. Defense trees for economic evaluation of security investments. In Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06), Vienna, Austria, 20–22 April 2006; pp. 8–423. [CrossRef]
11. Bistarelli, S.; Peretti, P.; Trubitsyna, I. Analyzing Security Scenarios Using Defence Trees and Answer Set Programming. *Electron. Notes Theor. Comput. Sci.* **2008**, *197*, 121–129. [CrossRef]
12. Bistarelli, S.; Fioravanti, F.; Peretti, P.; Santini, F. Evaluation of complex security scenarios using defense trees and economic indexes. *J. Exp. Theor. Artif. Intell.* **2012**, *24*, 161–192. [CrossRef]
13. Baca, D.; Petersen, K. Prioritizing Countermeasures through the Countermeasure Method for Software Security (CM-Sec). In Proceedings of the 11th International Conference, PROFES 2010, Limerick, Ireland, 21–23 June 2010; Ali Babar, M., Vierimaa, M., Oivo, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 176–190.
14. Roy, A.; Kim, D.S.; Trivedi, K.S. Attack Countermeasure Trees (ACT): Towards Unifying the Constructs of Attack and Defense Trees. *Sec. Commun. Netw.* **2012**, *5*, 929–943. [CrossRef]
15. Gadyatskaya, O. How to Generate Security Cameras: Towards Defence Generation for Socio-Technical Systems. In Proceedings of the Third International Workshop, GraMSec 2016, Lisbon, Portugal, 27 June 2016; Mauw, S., Kordy, B., Jajodia, S., Eds.; Springer: Cham, Switzerland, 2016; pp. 50–65.
16. Mauw, S.; Oostdijk, M. Foundations of Attack Trees. In Proceedings of the Information Security and Cryptology—ICISC 2005, Seoul, Republic of Korea, 1–2 December 2005; Won, D.H., Kim, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 186–198.
17. Pinchinat, S.; Acher, M.; Vojtisek, D. Towards Synthesis of Attack Trees for Supporting Computer-Aided Risk Analysis. In Proceedings of the SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, 1–2 September 2014.
18. Paul, S. Towards Automating the Construction & Maintenance of Attack Trees: A Feasibility Study. In Proceedings of the First International Workshop on Graphical Models for Security, GraMSec 2014, Grenoble, France, 12 April 2014; Kordy, B., Mauw, S., Pieters, W., Eds.; Volume 148, pp. 31–46. [CrossRef]
19. Pinchinat, S.; Acher, M.; Vojtisek, D. ATSyRa: An Integrated Environment for Synthesizing Attack Trees. In Proceedings of the Second International Workshop on Graphical Models for Security (GraMSec'15), Verona, Italy, 13 July 2015.
20. Bryans, J.; Nguyen, H.N.; Shaikh, S.A. Attack Defense Trees with Sequential Conjunction. In Proceedings of the 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE), Hangzhou, China, 3–5 January 2019; pp. 247–252. [CrossRef]
21. Wideł, W.; Audinot, M.; Fila, B.; Pinchinat, S. Beyond 2014: Formal methods for attack tree-based security modeling. *ACM Comput. Surv.* **2019**, *52*, 75. [CrossRef]
22. Horne, R.; Mauw, S.; Tiu, A. Semantics for specialising attack trees based on linear logic. *Fundam. Inform.* **2017**, *153*, 57–86. [CrossRef]
23. Audinot, M.; Pinchinat, S.; Kordy, B. Is My Attack Tree Correct? In Proceedings of the 22nd European Symposium on Research in Computer Security, Oslo, Norway, 11–15 September 2017; Foley, S.N., Gollmann, D., Snekkenes, E., Eds.; Springer: Cham, Switzerland, 2017; pp. 83–102.
24. Audinot, M. Assisted Design and Analysis of Attack Trees. Ph.D. Thesis, University Rennes 1, Rennes, France, 2018.
25. ATSyRA Studio. Available online: http://atsyra2.irisa.fr/ (accessed on 1 July 2023).
26. Vigo, R.; Nielson, F.; Nielson, H.R. Automated Generation of Attack Trees. In Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium, Vienna, Austria, 19–22 July 2014; pp. 337–350. [CrossRef]
27. Vigo, R. Nielson, F.; Nielson, H. Discovering, quantifying, and displaying attacks. *Log. Methods Comput. Sci.* **2016**, *12*.
28. Nielson, H.R.; Nielson, F.; Vigo, R. A Calculus for Quality. In Proceedings of the 9th International Symposium, FACS 2012, Mountain View, CA, USA, 11–13 September 2012; Păsăreanu, C.S., Salaün, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 188–204.
29. Probst, C.W.; Willemson, J.; Pieters, W. The attack navigator. *GraMSec 2015 (LNCS)* **2015**, *9390*, 1–17.
30. Sowka, K.; Cheah, M.; Doan, T.; Nguyen, H.; Shaikh, S. Towards Generation of Attack Trees Using Machine Learning. 2021. Available online: https://pure.coventry.ac.uk/ws/portalfiles/portal/53429439/Towards_Generation_of_Attack_Trees.pdf (accessed on 1 July 2023).
31. Mahmood, S.; Nguyen, H.N.; Shaikh, S.A. Systematic threat assessment and security testing of automotive over-the-air (OTA) updates. *Veh. Commun.* **2022**, *35*, 100468. [CrossRef]
32. Cheah, M.; Shaikh, S.A.; Bryans, J.; Wooderson, P. Building an automotive security assurance case using systematic security evaluations. *Comput. Secur.* **2018**, *77*, 360–379. [CrossRef]
33. Lallie, H.S.; Debattista, K.; Bal, J. An Empirical Evaluation of the Effectiveness of Attack Graphs and Fault Trees in Cyber-Attack Perception. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 1110–1122. [CrossRef]
34. Hong, J.B.; Kim, D.S.; Takaoka, T. Scalable Attack Representation Model Using Logic Reduction Techniques. In Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, Melbourne, Australia, 16–18 July 2013; pp. 404–411. [CrossRef]
35. Gadyatskaya, O.; Jhawar, R.; Mauw, S.; Trujillo-Rasua, R.; Willemse, T.A.C. Refinement-Aware Generation of Attack Trees. In Proceedings of the 13th International Workshop, STM 2017, Oslo, Norway, 14–15 September 2017; Livraga, G., Mitchell, C., Eds.; Springer: Cham, Switzerland, 2017; pp. 164–179.

36. Chulp, S.; Christl, K.; Schmittner, C.; Shaaban, A.M.; Schauer, S.; Latzenhofer, M. THREATGET: Towards Automated Attack Tree Analysis for Automotive Cybersecurity. *Information* **2023**, *14*, 28.
37. Gadyatskaya, O.; Mauw, S. Attack Tree Series: A case for dynamic attack tree analysis. In Proceedings of the 6th International Workshop, GraMSec 2019, Hoboken, NJ, USA, 24 June 2019.
38. Ali, A.T.; Gruska, D.P. Attack Trees with Time Constraints. In Proceedings of the 29th International Workshop on Concurrency, Specification and Programming (CS&P 2021), Berlin, Germany, 27–28 September 2021.
39. Jhawar, R.; Lounis, K.; Mauw, S.; Ramírez-Cruz, Y. Semi-automatically Augmenting Attack Trees Using an Annotated Attack Tree Library. In *Security and Trust Management*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2018; Volume 11091, pp. 85–101. [CrossRef]
40. CAPEC—Common Attack Pattern Enumeration and Classification. Available online: https://capec.mitre.org/ (accessed on 1 July 2023).
41. Mantel, H.; Probst, C.W. On the meaning and purpose of attack trees. In Proceedings of the 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), Hoboken, NJ, USA, 25–28 June 2019; pp. 184–199. [CrossRef]
42. Pinchinat, S.; Fila, B.; Wacheux, F.; Thierry-Mieg, Y. Attack Trees: A Notion of Missing Attacks. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer International Publishing: Berlin/Heidelberg, Germany, 2019; Volume 11720. [CrossRef]
43. Bryans, J.; Liew, L.S.; Nguyen, H.N.; Sabaliauskaite, G.; Shaikh, S.; Zhou, F. A Template-Based Method for the Generation of Attack Trees. In Proceedings of the 13th IFIP WG 11.2 International Conference, WISTP 2019, Paris, France, 11–12 December 2019; Laurent, M., Giannetsos, T., Eds.; Springer: Cham, Switzerland, 2020; pp. 155–165.
44. Sheyner, O.; Haines, J.; Jha, S.; Lippmann, R.; Wing, J. Automated generation and analysis of attack graphs. In Proceedings of the 2002 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 12–15 May 2002; pp. 273–284. [CrossRef]
45. Sheyner, O.; Wing, J. Tools for Generating and Analyzing Attack Graphs. In Proceedings of the Second International Symposium, FMCO 2003, Leiden, The Netherlands, 4–7 November 2003; de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 344–371.
46. Ivanova, M.G.; Probst, C.W.; Hansen, R.R.; Kammüller, F. Attack Tree Generation by Policy Invalidation. In Proceedings of the 9th IFIP WG 11.2 International Conference, WISTP 2015, Heraklion, Greece, 24–25 August 2015; Akram, R.N., Jajodia, S., Eds.; Springer: Cham, Switzerland, 2015; pp. 249–259.
47. Xu, J.; Venkatasubramanian, K.K.; Sfyrla, V. A methodology for systematic attack trees generation for interoperable medical devices. In Proceedings of the 2016 Annual IEEE Systems Conference (SysCon), Orlando, FL, USA, 18–21 April 2016; pp. 1–7. [CrossRef]
48. Santra, S. *Semi-Automated Generation of Networked Vulnerability-Attack Countermeasure Trees for Security Analysis*; University of Canterbury: Christchurch, New Zealand, 2017.
49. Falco, G.; Viswanathan, A.; Caldera, C.; Shrobe, H. A Master Attack Methodology for an AI-Based Automated Attack Planner for Smart Cities. *IEEE Access* **2018**, *6*, 48360–48373. [CrossRef]
50. Cheah, M.; Nguyen, H.; Bryans, J.; Shaikh, S.A. Formalising Systematic Security Evaluations Using Attack Trees for Automotive Applications. In Proceedings of the 11th IFIP WG 11.2 International Conference, WISTP 2017, Heraklion, Greece, 28–29 September 2017; Springer: Berlin/Heidelberg, Germany, 2018; pp. 113–129. [CrossRef]
51. Roscoe, A.W. *Understanding Concurrent Systems*; Springer: Berlin/Heidelberg, Germany, 2010.
52. FDR4. Available online: https://www.cs.ox.ac.uk/projects/fdr/ (accessed on 5 September 2018).
53. Gadyatskaya, O.; Hansen, R.R.; Larsen, K.G.; Legay, A.; Olesen, M.C.; Poulsen, D.B. Modelling Attack-Defense Trees Using Timed Automata. In *Formal Modeling and Analysis of Timed Systems*; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9884, pp. 35–50. [CrossRef]
54. Kordy, B.; Mauw, S.; Radomirovic, S.; Schweitzer, P. Attack-Defense Trees. *J. Log. Comput.* **2014**, *24*, 55–87. [CrossRef]
55. Cobos, L.P.; Ruddle, A.R.; Sabaliauskaite, G. Cybersecurity Assurance Challenges for Future Connected and Automated Vehicles. In Proceedings of the 31st European Safety and Reliability Conference, Angers, France, 19–23 September 2021; pp. 2038–2045. [CrossRef]
56. Kumar, R.; Stoelinga, M. Quantitative Security and Safety Analysis with Attack-Fault Trees. In Proceedings of the 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE), Singapore, 12–14 January 2017; pp. 25–32. [CrossRef]
57. André, É.; Lime, D.; Ramparison, M.; Stoelinga, M. Parametric Analyses of Attack-Fault Trees. In Proceedings of the 2019 19th International Conference on Application of Concurrency to System Design, ACSD 2019, Aachen, Germany, 23–28 June 2019; pp. 33–42. [CrossRef]
58. CWE—Common Weakness Enumeration. Available online: https://cwe.mitre.org/ (accessed on 1 July 2024).
59. NVD—National Vulnerability Database. Available online: https://nvd.nist.gov/ (accessed on 1 July 2024).