Attack Defense Trees with Sequential Conjunction

Nguyen, H. N., Bryans, J. & Shaikh, S.

Original citation & hyperlink:
Nguyen, HN, Bryans, J & Shaikh, S 2019, Attack Defense Trees with Sequential Conjunction. in 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE). Proceedings / IEEE International Symposium on High-Assurance Systems Engineering. IEEE International Symposium on High-Assurance Systems Engineering (IEEE, pp. 247-252, Workshop on Security issues in Cyber-Physical System(SecCPS), In conjunction with IEEE HASE, Hangzhou, China, 3/01/19. https://dx.doi.org/10.1109/HASE.2019.00045

DOI 10.1109/HASE.2019.00045
ISSN 1530-2059
ESSN 2640-7507

Publisher: IEEE

Attack Defense Trees with Sequential Conjunction

Jeremy Bryans Hoang Nga Nguyen Siraj Ahmed Shaikh
Systems Security Group, Institute for Future Transport and Cities (FTC)
Coventry University, Coventry CV1 5FB, UK
{jeremy.bryans,hoang.nguyen,siraj.shaikh}@coventry.ac.uk

***Abstract*—Attack defense trees are used to show the interaction between potential attacks on a system and the system defenses. In this paper we present a formal semantic model for attack defense trees with sequential composition, allowing for the description of attacks that are performed as a sequence of steps. The main contributions of our work are a formal representation of attack defense trees with sequential conjunction, a demonstration that this representation is equivalent to a process-algebraic one, and an algorithm for identifying the existence of attacks. We illustrate with an attack on over the air updates.**

***Index Terms*—Cyber security, Attack defense trees, Formal semantics.**

I. Introduction

The security of all large, complex systems evolves, and often comes to public attention as attacks on those systems are demonstrated. This is exemplified within the automotive industry, where the diverse technologies employed by vehicles have led to a series of high-profile attacks [2], [4], [11].

Within the automotive industry, ISO 26262 [6] is the standard for functional safety of the vehicle electrical and electronic systems, but it does not address cybersecurity[1]. J3061 [15] provides a set of high-level guiding principles for a vehicle manufacturer seeking to take account of cybersecurity, and provides information on tools and techniques.

One of the techniques available is an analysis based on *attack trees*. Attack trees are a systematic approach to categorise and evaluate the logical paths that an attacker could take through the system to lead to an attack. They were popularised by Schneier in [16] and given a formal interpretation in [10]. This formal semantics was later extended with sequential conjunction in [7], in order to allow for attacks in which the order of execution of attack steps was significant.

A limitation with attack trees is their inability to capture the interaction between the attacks and the defenses of a system. This observation led to the development of attack defense trees (ADTs), which were given a formal interpretation in [9].

In previous work [1] we presented a method for systematically generating tests based on attack trees. The approach, which was inspired by model based testing, was designed for the vehicle industry in which black box components are the norm. It therefore allowed for the possibility that the system under test was not fully specified.

In this paper we consider attack defense trees that include sequential conjunction. The main contributions of this work are a formal representation of attack defense trees with sequential conjunction, a demonstration that this representation is equivalent to a process-algebraic one, and an algorithm for identifying the existence of attacks.

[1]Cybersecurity is an area of consideration for ISO 26262 version 2, which is under development.

II. Preliminaries

Below we define attack trees as series-parallel graphs, and introduce the process algebra Communicating Sequential Processes [14] (CSP) and its trace semantics.

A. 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**) [7]. We follow the formalisation of attack trees given in [7], [10]. If $\mathbb{A}$ is the set of 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 [7]. The definition of SP graphs requires first the definition of source-sink graphs and here we use the definitions from [7].

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 $s \neq z$.*

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'. 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: $G \parallel G' = (V \setminus \{s, z\} \dot\cup V', E^{[s'/s, z'/z]} \dot\cup E', s', z')$.

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 [1], 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 II-B) means that we can use synonymous operators to transform a pre-built attack tree.

B. CSP

A brief overview of the subset of CSP that we use in this paper is given here. A more complete introduction may be found in [14]. Given a set of events Σ, CSP processes are defined by the following syntax:

$$P \quad ::= \quad Stop \mid e \to P \mid P_1 \,\Box\, P_2 \mid P_1;\ P_2 \mid P_1 \underset{A}{\parallel} P_2$$

where $e \in \Sigma$, $A, B \subseteq \text{events}$. 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 $\checkmark \to Stop$. The process $e \to P$ engages in the event e then behaves as P. The choice $P_1 \,\Box\, P_2$ 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 $P_1 \underset{A}{\parallel} P_2$ requires P_1 and P_2 to synchronise on events in $A \cup \{\checkmark\}$. All other events are executed independently, and $P_1 \,|||\, P_2$ is an abbreviation for $P_1 \underset{\emptyset}{\parallel} P_2$.

There are different semantics models for CSP processes [14]. For the purpose of this paper, 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:

- $traces(Stop) = \{\langle\rangle\}$;
- $traces(e \to P) = \{\langle\rangle\} \cup \{\langle e\rangle \frown tr \mid tr \in traces(P)\}$;
- $traces(P_1 \,\Box\, P_2) = traces(P_1) \cup traces(P_2)$;
- $traces(P_1;\ P_2) = traces(P_1) \cap \Sigma^* \cup \{tr_1 \frown tr_2 \mid tr_1 \frown \langle\checkmark\rangle \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$;
- $traces(P_1 \underset{A}{\parallel} P_2) = \{tr \in tr_1 \underset{A}{\parallel} tr_2 \mid tr_1 \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$

  where $tr_1 \underset{A}{\parallel} tr_2 = tr_2 \underset{A}{\parallel} tr_1$ is defined as follows with $a, a' \in A \cup \{\checkmark\}$ and $b, b' \notin A$:
  - $\langle\rangle \underset{A}{\parallel} \langle\rangle = \{\langle\rangle\}$; $\langle\rangle \underset{A}{\parallel} \langle a\rangle = \emptyset$; $\langle\rangle \underset{A}{\parallel} \langle b\rangle = \{\langle b\rangle\}$;
  - $\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle b\rangle \frown tr_2 = \{\langle b\rangle \frown tr \mid tr \in \langle a\rangle \frown tr_1 \underset{A}{\parallel} tr_2\}$;
  - $\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle a\rangle \frown tr_2 = \{\langle a\rangle \frown tr \mid tr \in tr_1 \underset{A}{\parallel} tr_2\}$
  - $\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle a'\rangle \frown tr_2 = \emptyset$ where $a \neq a'$;
  - $\langle b\rangle \frown tr_1 \underset{A}{\parallel} \langle b'\rangle \frown tr_2 = \{\langle b\rangle \frown tr \mid tr \in tr_1 \underset{A}{\parallel} \langle b'\rangle \frown tr_2\} \cup \{\langle b'\rangle \frown tr \mid tr \in \langle b\rangle \frown tr_1 \underset{A}{\parallel} tr_2\}$

As mentioned earlier, trace interleaving is defined as:

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

For convenience, we sometimes use the notion of interleaving and concatenation over two set of traces. In particular, given two sets S_1 and S_2 of traces, $S_1 \,|||\, S_2 = \bigcup_{s_1 \in S_1, s_2 \in S_2} s_1 \,|||\, s_2$ 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 [3] for checking trace-refinements.

In [1], 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 SP graph 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:

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

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

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

III. ADTs with Sequential Conjunction

The above attack defense trees (ADTs) allow us to capture the interaction between attack and defense, and therefore between attacker and defender. Now we extend ADTs to include sequential composition, 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 a new operator **SAND**, or sequential **AND**.

A. 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:

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

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 [5], our main interest is to check if there exists a successful attack (defense) in a given attack(defense)-rooted tree. In this paper, it is called *ADS tree checking decision problem* and 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.

B. Semantics

The first model is a natural extension of semantics for ADS trees using Series-Parallel graphs (SPGs). SPGs 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.

1) SPG semantics: We extend the Series-Parallel Graph (SPG) semantics of attack trees with sequential operator. Particularly, ADS trees are interpreted as sets of SP graphs. Each such set captures possible attacks (defenses, resp.) and is associated with a set of SP graphs describing possible defenses (attacks, resp.). 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 (defenses, respectively); 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 **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

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

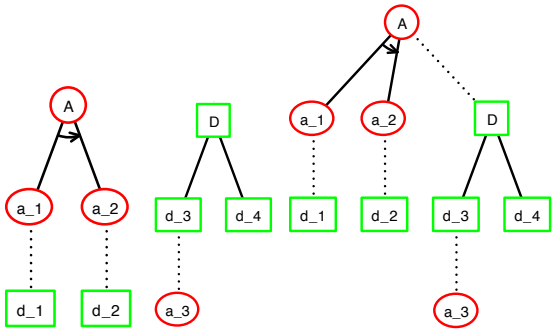
However, each of such G_2 can be countered again by the corresponding P_2, i.e., $(G_2, P_2) \in [\![t_2]\!]_{SP}$, if $P_2 \neq \emptyset$. Then attacks in G_1 are combined with those from P_2. They can only be countered by defenses without countermeasures in $[\![t_2]\!]_{SP}$, i.e., those in G'_2 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 an example of the last case, consider the ADS tree in Fig. 1(a), 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 consider the defense mechanism in Fig. 1(b). It is made up of two components: d_3, and d_4. There is a counter attack against d_3 (a_3) but not against d_4. The SPG semantics of this graph is given by $\{(\{\xrightarrow{d_3}\}, \{\xrightarrow{a_3}\}), (\{\xrightarrow{d_4}\}, \emptyset)\}$.

Now consider Fig. 1(c). In this, the constructed defense from Fig. 1(b) has been added to the attack tree, and the whole attack defense tree is given by $(A$ **C** $D)$. The additional defense mechanism D is capable of defending against the sequential attack a_1 followed by a_2. The SPG semantics of the overall



(a) Attack. (b) Defense. (c) Defense in situ.

Fig. 1: Attack defense trees.

tree is 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 any of defense actions $\xrightarrow{d_1}$, $\xrightarrow{d_2}$ or $\xrightarrow{d_4}$ is sufficient to defend against either of the two options open to the attacker.

Given the semantics of ADS trees, let us formalise the analysis question of interest. Given an ADS tree t, it is successful if and only if there exists a pair $(G, \emptyset) \in [\![t]\!]_{SP}$, i.e., attacks or defenses captured in G has no corresponding countermeasures.

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, let us extend the function $serials(\cdot)$, presented in [1], to accommodate outcomes 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 define 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 $b \in \mathbb{B}$;
- $[\![t_1 \textbf{ OR } t_2]\!]_T = [\![t_1]\!]_T \cup [\![t_2]\!]_T$;
- $[\![t_1 \textbf{ AND } t_2]\!]_T = \{(S_1 \,|||\, S_2, T_1 \cup T_2) \mid (S_1, S_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T\}$;
- $[\![t_1 \textbf{ SAND } t_2]\!]_T = \{(S_1 \frown S_2, T_1 \cup T_2) \mid (S_1, T_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T\}$;
- $[\![t_1 \textbf{ C } t_2]\!]_T =$
  $\{(S_1, T_1 \cup (\bigcup_{(S_2,T_2) \in [\![t_2]\!]_T} S_2)) \mid (S_1, T_1) \in [\![t_1]\!]_T\} \cup$
  $\{(S_1 \,|||\, T_2, T_1 \cup (\bigcup_{(S_2',\emptyset) \in [\![t_2]\!]_T} S_2')) \mid (S_1, T_1) \in [\![t_1]\!]_T, (S_2, T_2) \in [\![t_2]\!]_T, T_2 \neq \emptyset\}$.

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

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

The following result is immediate.

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

This means to check if a tree t is successful, it is sufficient to check the existence of $(S, \emptyset) \in [\![t]\!]_T$.

IV. 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 [3], a model checker for CSP, is employed to elicit a successful attack (defense).

A. 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 [9] 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. The following correctness result is immediate.

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 = t1 OR t2 then
        return check(t1) ∨ check(t2);
    else if t = t1 AND t2 or t = t1 SAND t2 then
        return check(t1) ∧ check(t2);
    else if t = t1 C t2 then
        return check(t1) ∧ ¬check(t2);
    end
end
```

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

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.

B. 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:

$trans(b) = \{(b \to Skip, Stop, \top)\}$
$trans(t_1 \textbf{ OR } t_2) = trans(t_1) \cup trans(t_2)$;
$trans(t_1 \textbf{ AND } t_2) = \{(P_1 \,|||\, P_2, Q_1 \,\Box\, Q_2, O_1 \wedge O_2) \mid (P_1, Q_1, O_1) \in trans(t_1), (P_2, Q_2, O_2) \in trans(t_2)\}$;
$trans(t_1 \textbf{ SAND } t_2)\{(P_1;\ P_2, Q_1 \,\Box\, Q_2, O_1 \wedge O_2) \mid (P_1, Q_1, O_1) \in trans(t_1), (P_2, Q_2, O_2) \in trans(t_2)\}$;
$trans(t_1 \textbf{ C } t_2) = \{(P_1, Q_1 \,\Box\, (\Box_{(P_2,Q_2,O_2) \in trans(t_2)} P_2), \bot) \mid (P_1, Q_1, O_1) \in trans(t_1)\}$

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

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 attack-rooted tree t. If t is defense-rooted, the same argument is also applied where "attack" is exchanged with "defense". Given such a pair $(P, Q, O) \in trans(t)$, 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 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 $t = t_1$ **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 $(P_1 \,|||\, P_2, Q_1 \,\Box\, Q_2, O_1 \wedge O_2)$ where $(P_i, Q_i, O_i) \in trans(t_i)$ for $i \in \{1, 2\}$. The same explanation applies for $t = t_1$ **SAND** t_2 by replacing $|||$ 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 $(P_1, Q_1 \,\Box\, (\Box_{(P_2,Q_2,O_2)\in trans(t_2)} P_2), \bot)$ 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 $(P_1 \,|||\, Q_2, Q_1 \,\Box\, (\Box_{(P_3,Q_3,\top)\in trans(t_2)}), O)$ 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. $\forall t \in \mathbb{T}_{ADS} : traces(trans(t)) = [\![t]\!]_T$.

C. 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. Out 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-termniated traces is captured by $Run(\mathbb{A})$ if t is attack-rooted or $Run(\mathbb{A})$ 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 of the $(P, Q, \top)$. Therefore, t is not successful. Conversely, if the refinement fails, i.e., there exists a counter example, t is successful. By using FDR, this refinement can be automatically checked. If it fails, a counter example will be produced which is an evidence of an attack (or defense if t is defense-rooted) without countermeasures.

V. Case study

We extend ADTool2 [8] with ADS trees[2]. ADTool2 supports graphically modelling and analysing attack trees with SAND and attack defense trees without SAND. The extension also accommodates the translation to CSP. To illustrate the use of this tool extension, we introduce the following case study.

Traditional telematics capability for vehicles have evolved to support critical functionality including firmware updates for on-board Electronic Control Units (ECUs). Known as Over-The-Air (OTA) updates, this is significant to remotely address feature updates and performance flaws; indeed Tesla demonstrated this to address problems with braking systems on their recent model [20]. Supporting safety-critical functions through such connectivity however brings to fore the concerns around security of such components [13].

A key concern here is remote exploitation of vulnerabilities on the communication units on-board vehicles. Miller and Valasek brought attention to this [19] when they managed to perform remote code execution due to authentication flaws in the Uconnect System [18]. A further concern are attacks due to infiltration of software supply chains [12]. Essentially, third party software could carry a 'backdoor' to bypass authentication measures.

Traditional measures to detect intrusions over communication channels have been extended over automotive Controller Area Network (CAN) [17]. Signature-based methods are constrained by the knowledge of known exploits, whereas anomaly detection systems pose the usual challenge of accurately distinguishing between normal and abnormal activity at an acceptable rate.

The concerns regarding OTA are depicted in a ADS tree. It is modelled in the extension of ADTool2, depicted in Figure 2, together with identified counter measures as defenses. The

[2]A pre-release is downloadable from http://goo.gl/Ebkb8i.

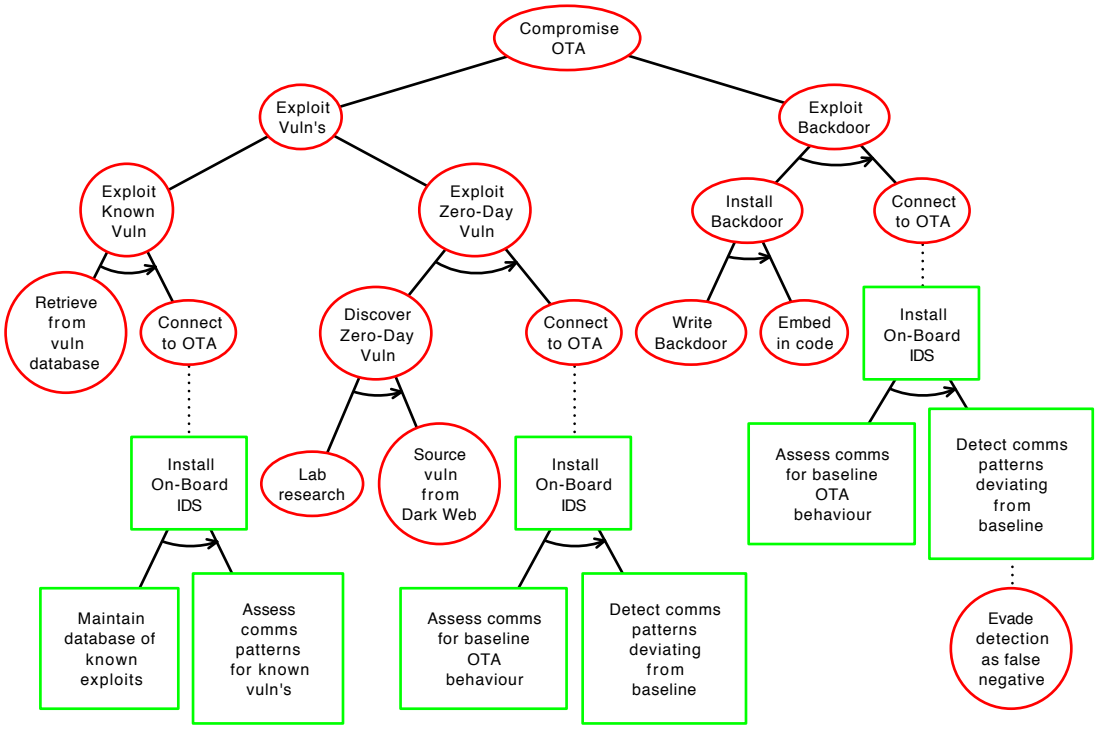


Fig. 2: The ADS shows compromise of Over-The-Air (OTA) feature as the top objective. The route to such compromise is either an existing or zero-day vulnerability (present in protocol or interface design or implementation), or a dedicated backdoor. Mitigation includes On-Board IDS, either signature-based or detecting anomalies, the latter of which is open to false negatives evading detection of true intrusion attempts. The use of the **SAND** operator depicts the sequential order for the relevant steps.

tool then is used to generate the CSP translation, which is fed into FDR to check the refinement mentioned in Section IV. FDR confirms that the refinement fails with a counter example $\langle$*Write Backdoor, Embed in Code, Evade Detection as False Negative, Connect to OTA*$\rangle$, which turns out to be the only possible attack in the ADS tree.

VI. Conclusion and Future Work

This work has given a formal representation of attack defense trees with sequential conjunction, a demonstration that this representation is equivalent to a process-algebraic one, an algorithm for identifying the existence of attacks, and an example featuring an attack on OTA updates. In further work we will explore further the application to automotive security.

References

[1] M. Cheah, HN. Nguyen, J. Bryans, and S. A. Shaikh. Formalising Systematic Security Evaluations Using Attack Trees for Automotive Applications. In *WISTP'17*, pages 113–129. Springer, 2018.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, Karl K., A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. *System*, 2011.

[3] FDR4. Available at https://www.cs.ox.ac.uk/projects/fdr/, 2018-09-05.

[4] I. Foster, A. Prudhomme, K. Koscher, and S. Savage. Fast and Vulnerable: A Story of Telematic Failures. In *WOOT'15*, 2015.

[5] O. Gadyatskaya, R. R. Hansen, K. G. Larsen, A. Legay, M. Chr. Olesen, and D. B. Poulsen. Modelling Attack-defense Trees Using Timed Automata. In *Formal Modeling and Analysis of Timed Systems*, volume 9884, pages 35–50. Springer, 2016.

[6] ISO. ISO26262: Road Vehicles – Functional Safety, 2011.

[7] R. Jhawar, B. Kordy, S. Mauw, S. Radomirovi, and R. Trujillo-Rasua. Attack Trees with Sequential Conjunction. In *ICT Systems Security and Privacy Protection*, volume 455, pages 339–353. Springer, 2015.

[8] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer. ADTool: Security Analysis with Attack–Defense Trees. In *Quantitative Evaluation of Systems*, volume 8054, pages 173–176. Springer, 2013.

[9] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Attack-defense trees. 24(1):55–87, 2014-02-01.

[10] S. Mauw and M. Oostdijk. Foundations of Attack Trees. In *ICISC 2005*, volume 3935, pages 186–198. Springer, 2006.

[11] C. Miller and C. Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *Defcon 23*, 2015:1–91, 2015.

[12] National Cyber Security Centre (NCSC). Example Supply Chain Attacks: Third party software providers. Available at https://www.ncsc.gov.uk/guidance/example-supply-chain-attacks. Accessed: 2018-10-20.

[13] D. K. Nilsson, U. E. Larson, and E. Jonsson. Creating a secure infrastructure for wireless diagnostics and software updates in vehicles. In *Computer Safety, Reliability, and Security*. Springer, 2008.

[14] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[15] SAE. J3061 – Cybersecurity Guidebook for CP Vehicle Systems, 2016.

[16] B. Schneier. AT: Modeling Security Threats. *Dr. Dobb's Journal*, 1999.

[17] A. Tomlinson, J. Bryans, and S. A. Shaikh. Towards viable intrusion detection methods for the automotive controller area network. 2nd ACM Computer Science in Cars Symposium, Munich, Germany.

[18] US-ICS-CERT. Harman-kardon uconnect vulnerability. advisory (icsa-15-260-01). Available at https://ics-cert.us-cert.gov/advisories/ICSA-15-260-01. Accessed: 2018-10-20.

[19] C. Valasek and C. Miller. A survey of remote automotive attack surfaces. Available at https://ioactive.com/pdfs/IOActive_Remote_Attack_Surfaces.pdf. Accessed: 2018-10-20.

[20] WIRED. Tesla's Quick Fix for its Braking System came from the Ether. Available at https://www.wired.com/story/tesla-model3-braking-software-update-consumer-reports/. Accessed: 2018-10-20.