Formalising Systematic Security Evaluations Using Attack Trees for Automotive Applications

Madeline Cheah, Hoang Nga Nguyen, Jeremy Bryans, Siraj A. Shaikh

Formalising Systematic Security Evaluations using Attack Trees for Automotive Applications

Madeline Cheah✉, Hoang Nga Nguyen, Jeremy Bryans, and Siraj A. Shaikh

Centre for Mobility and Transport Research, Coventry University, Coventry, UK
{cheahh2,hoang.nguyen,jeremy.bryans,siraj.shaikh}@coventry.ac.uk

Abstract. Vehicles are insecure. To protect such systems, we must begin by identifying any weaknesses. One approach is to apply a systematic security evaluation to the system under test. In this paper we present a method for systematically generating tests based on attack trees. We formalise the attack trees as provably-equivalent process-algebraic processes, then automatically generate tests from the process-algebraic representation. Attack trees may include manual input, (and thus so will some test cases) but scriptable test cases are automatically executed. Our approach is inspired by model based testing, but allows for the fact that we do not have a specification of the system under test. We demonstrate this methodology on a case study and find that this is a viable method for automation of systematic security evaluations.

Keywords: automotive security · attack trees · secure design · security testing · Bluetooth

1 Introduction

Vehicles are extremely reliant on software and connectivity to enable the functionality desired by customers. The explosion of diverse technologies in cars has meant that cybersecurity of vehicles has become a mainstream concern for manufacturers; such concerns arise out of a number of attacks on components [4,14], internal vehicular network [5,12] and through external interfaces [4,10]. Systematic and automated testing to assure against many such attacks is a challenge, and an ad-hoc approach (i.e. a subjective prioritisation of when and where to test) usually means there is less than optimal coverage of vulnerabilities.

The contribution of this paper is a well-founded methodology to systematically evaluate the cybersecurity of a vehicle, using a model checker and a translation of attack trees into a process algebra. The methodology is inspired by model-based security testing.

The rest of the paper is organised as follows. In Section 2, we discuss related work. Section 3 contains the semantics of attack trees as source-sink graphs and the traces model of CSP [20]. This is followed by an overview of our methodology in Section 4, including a translation function for attack trees into the traces model of CSP, and a demonstration of the equivalence of the two models. We then discuss the implementation of this methodology (Section 5) and apply it to

a case study involving diagnostic devices that attach to the vehicular on-board diagnostics port (Section 6). Section 7 concludes the paper.

2 Related Work

We discuss the attack tree formalism in an automotive context in Section 2.1 and model based security testing in Section 2.2.

2.1 Attack Trees

The foundations of formal descriptions of attacks were laid by [15] and in particular their process algebraic nature was recognised in [28]. Automotive specific attack trees have also been discussed in literature where [24] looked at attack tree generation and gave formal descriptions of the trees. This is orthogonal to our research (in translating low level attack trees) as in our case the trees have already been pre-built based on reconnaissance of a black box system, rather than the automatic generation of an attack tree from a fully specified system-under-test (SUT).

Attack tree generation is still challenging as the SUT has unknown specifications (black box). Other examples of attack trees in the automotive context are those as described in the 'E-safety vehicle intrusion protected applications' (EVITA) project [21] and the SAE J3061 Cybersecurity Guidebook for Cyber-Physical Vehicle Systems [23], although these are informal. Application of executable attack trees has been demonstrated [3], however, automation was limited to test case execution. The attack trees were informally described and manually created.

2.2 Model-based Security Testing

Model-based security testing (MBST) is a special case of model-based testing (MBT) with a focus on security requirements. One can find a succinct classification of different approaches with respect to MBT and MBST in [8, 27], where MBT is comprised of the following: (1) A formal model of the SUT is collected usually from specification documents as the result of the design phase of the SUT, or constructed from system requirements. This model can be at different abstraction levels where the most abstract one contains all possible behaviours; (2) Test case filter criteria are then determined. They are usual informal descriptions capturing a subset of behaviours of the formal model which can be drawn from the requirements or the structure of the formal model (e.g., state coverage, transition coverage). In terms of MBST, these criteria concentrate on system security such as security properties (e.g.,confidentiality, integrity and availability), security mechanisms, and the environment (e.g., attack strategies). (3) These criteria are then formalised into test case specifications, usually compatible with the formal model. (4) Given the formal model of the SUT and the test case specifications, test cases are automatically generated by different technologies such as

model checking, theorem proving, or graph search algorithms. Finally, (5) these test cases are executed on the SUT to provide verdicts. This involves translation of test cases from the abstract level of the formal model into concrete test scripts that are executable on SUT. Then, the execution result is translated back to the abstract level for comparison with the expected output of the generated test cases.

3 Semantic models

In this section we give the formal notation and source-sink semantics of attack trees (Section 3.1) and the notation and finite trace semantics of the formal language Communicating Sequential Process (CSP) (Section 3.2).

3.1 Attack Trees

Attack trees were first created to describe the actions of an attacker in an methodical manner [25].

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 are complete when all child nodes are carried out, and `OR` nodes are complete when at least one child node is complete.

Extensions have been proposed using `Sequential AND` (or `SAND`) [13]. There are two types of ordering: time dependent or condition dependent. In this paper we adopt the condition dependent paradigm.

We follow the formalisation of attack trees given in [13, 15]. If $\mathbb{A}$ is the set of possible atomic attacker actions, the elements of the attack tree $\mathbb{T}$ are $\mathbb{A} \cup \{OR, AND, SAND\}$, and an attack tree is generated by the following grammar, where $a \in \mathbb{A}$:

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

Attack tree semantics have been defined by interpreting the attack tree as a set of series-parallel (SP) graphs [13]. Definition of SP graphs requires first the definition of source-sink graphs and here we use the definitions from [13].

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:

$$[\![\cdot]\!]_{\mathcal{SP}} : \mathbb{T} \to \mathcal{P}(\mathbb{G}_{\mathcal{SP}})$$

This is defined recursively. If $a \in \mathbb{A}$, $t_i \in \mathbb{T}$, and $1 \leq i \leq k$, then

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

Since, in this paper, we base the construction of the attack tree on penetration testing techniques, all leaves on the tree can be considered actions. The combination of these actions can be translated into the processes that form part of a test case. This is conducive to the use of process algebra such as Communicating Sequential Processes (CSP) and furthermore 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

We give here a brief overview of the subset of CSP that we use in this paper. A more complete introduction may be found in [20].

Given a set of events Σ, CSP processes are defined by the following syntax:

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

where $e \in \Sigma$

and $A, B \subseteq$ events. For convenience, the set of CSP processes defined via the above syntax is denoted by CSP.

To mark the termination of a process, a special event ✓ is used.

In the above definition, the process *Stop* is the most basic one, which does not engage in any event and represents deadlock. In addition, *Skip* is an abbreviation for $\checkmark \to \mathit{Stop}$. It only exhibits ✓ and then behaves as *Stop*.

The prefix $e \to P$ specifies a process that is only willing to engage in the event e, then behaves as P. The external 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. Finally, the interleaving operator $P_1 \,|||\, P_2$ allows both P_1 and P_2 to execute concurrently and independently, except for $\checkmark$.

There are different semantics models for CSP processes [20]. 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 $\Sigma^{*\checkmark} = \{tr \frown en \mid tr \in \Sigma^* \wedge en \in \{\langle\rangle, \langle\checkmark\rangle\}\}$.

The trace tr_1 is a *prefix* of a trace tr_2, written as $tr_1 \leq tr_2$, iff $\exists\, tr' : tr_1 \frown tr' = tr_2$. Events in $A \subseteq \Sigma \cup \{\checkmark\}$ may be abstracted away from a trace tr by a hiding operator, written as $tr \setminus A$ and defined as

$$tr \setminus A = \begin{cases} \langle\rangle & \text{if } tr = \langle\rangle \\ \langle a\rangle \frown (tr' \setminus A) & \text{if } tr = \langle a\rangle \frown tr' \wedge a \notin A \\ tr' \setminus A & \text{if } tr = \langle a\rangle \frown tr' \wedge a \in A. \end{cases}$$

For convenience, when $A = \{a\}$, we shall simply write $tr \setminus a$. 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:

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

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

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.

4 Methodology

In this section we present an overview of the whole methodology (Figure 1) of our paper. The context of our work is the automotive industry. Vehicle manufacturers often incorporate off-the-shelf (OTS) components into their work, and their specifications are not always available. Manufacturers thus have to approach testing with this uncertainty.

We begin by assuming a System under Test (SUT), which may be either an OTS component or contain one. We also assume the existence of a corresponding attack tree (see Figure 1). Section 6 illustrates our approach with an attack tree developed for a vehicle network that includes a Bluetooth connection. It is worth observing here that an attack tree for Bluetooth systems essentially systematises the known attacks on Bluetooth, and therefore although it may be updated as new attacks are constructed, the development of the attack tree is a one-off cost. The same attack tree will work for any Automotive Bluetooth system.

If a specification (or abstract model) of the SUT is available we use it, but in many cases (including the example in Section 6) the specification of the SUT is confidential or contains confidential components. In this case we can under-approximate a specification to begin the process. Successive iterations of the process allow us to refine this under-approximation. We illustrate this under-approximation in Section 5. The approach is to generate tests against this model. Tests are automatically generated using FDR, the refinement checker for CSP. The specification and the attack tree are compared. Each possible route through the attack tree represents a potential attack, and the Test Case Generator compiles a list of all the attacks that the specification permits. Note that in the case of an under-approximated specification all possible attacks will be permitted.

The next step is to convert the formal tests into implementation tests. This process is detailed in Section 5.2. Note that these implementation tests are in fact attacks (or potential attacks) on the SUT. Not all the test implementation tests generated from an attack tree can be fully automated. The attack tree may contain nodes that require manual input, in which case the implementation tests will require (partial) manual interaction. The ones that do not require manual input may be executed directly on a Testbed. The report contains the test results. Since tests are really attacks on the SUT, we consider a test to be successful if the attack succeeds.

In the remainder of this section we present in more detail the transformation of attack trees to CSP processes (Section 4.1), as well as a proof of the equivalence of the semantic models.

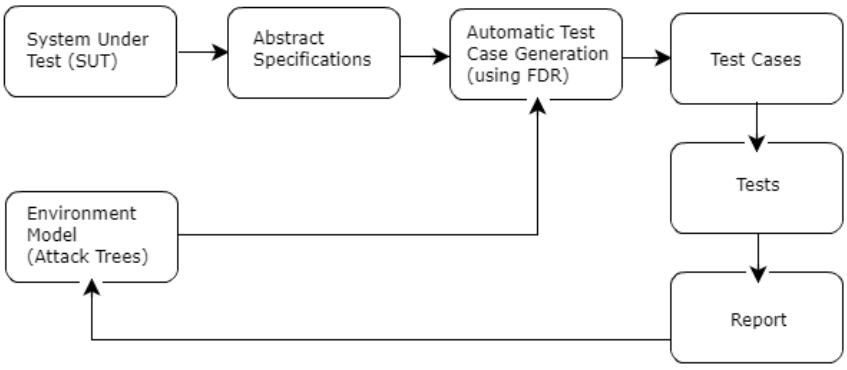

Fig. 1. Overview of our formal systematic security methodology

4.1 Transforming Attack Trees into CSP processes

In this section we use the process algebra CSP to describe the attack tree. We choose CSP because as a process algebra it is able to represent and combine the actions of the attack tree into a set of processes that could subsequently be used for test case generation.

In principle, the logic gates of the attack tree can be considered CSP operators [11] as follows:

- Since the `AND` logic gate demands that all actions must be successful for the branch to be considered complete, the interleave operator (|||) is used. This operator joins processes that operate concurrently but without them necessarily interacting or synchronising.
- The sequential composition operator (;) is used for the `SAND` logic gate. The former echoes the `SAND` logic gate, in that the first process must terminate successfully before the next is allowed;
- The external choice operator ($\Box$) (where any process could be chosen dependent on the environment in which it operates) is used for the `OR` logic gate.

Formally, we define the following transformation function trans : $\mathbb{T}_{SAND} \rightarrow CSP$ where $\Sigma = \mathbb{A}$:

- $\text{trans}(a) = a \rightarrow \mathit{Skip}$ for $a \in \mathbb{A}$;
- $\text{trans}(OR(t_1, \ldots, t_n)) = \text{trans}(t_1) \,\Box\, \ldots \,\Box\, \text{trans}(t_n)$;
- $\text{trans}(AND(t_1, \ldots, t_n)) = \text{trans}(t_1) \,|||\, \ldots \,|||\, \text{trans}(t_n)$;
- $\text{trans}(SAND(t_1, \ldots, t_n)) = \text{trans}(t_1);\ \ldots;\ \text{trans}(t_n)$;

In order to show the correctness of the above transformation, it is necessary to make the two semantics in $\mathbb{G}_{\mathcal{SP}}$ and $\Sigma^{*\checkmark}$ compatible for comparison. Recall from [13] that each SP graph represents a possible way to carry out an attack. In such a graph, an AND vertex indicates that actions along its branches must

be executed. However, there is no restriction on the order of their executions. In other words, their executions are interleaving in general. Therefore, it is possible to serialise the actions from a SP graph where parallel compositions of graphs is considered as interleaving and sequential composition as concatenation. Given G in $\mathbb{G}_{\mathcal{SP}}$, let serials($G$) denote the set of all possible ways to serialise G, which is formally defined as follow:

- serials($\xrightarrow{a}$) = $\{\langle a\rangle\}$;
- serials($G_1 \parallel G_2$) = $\{tr \in tr_1 \mid\mid\mid tr_2 \mid tr_1 \in \text{serials}(G_1) \wedge tr_2 \in \text{serials}(G_2)\}$;
- serials($G_1 \cdot G_2$) = $\{tr_1 \frown tr_2 \mid tr_1 \in \text{serials}(G_1) \wedge tr_2 \in \text{serials}(G_2)\}$.

For convenience, we denote the set of all prefixes from serials(G) by pserials(G), i.e., pserials(G) = $\{tr \mid \exists\, tr' \in \text{serials}(G) : tr \leq tr'\}$. We also denote pserials(t) = $\bigcup_{G \in [\![t]\!]_{\mathcal{SP}}} \text{pserials}(G)$ for all attack trees $t \in \mathbb{T}_{SAND}$.

The correctness of transforming attack trees into CSP processes is guaranteed by the following result:

Lemma 1. $\forall\, t \in \mathbb{T}_{SAND}$, pserials($t$) = *traces*(trans($t$)) $\setminus$ ✓.

Proof. The proof is done by induction on the structure of t.

Base case: Consider $t = a$; then, $[\![t]\!]_{\mathcal{SP}} = \{\xrightarrow{a}\}$ and pserials(t) = $\{\langle\rangle, \langle a\rangle\}$. We also have trans($t$) = $a \rightarrow \mathit{Skip}$ and *traces*(trans(t)) = $\{\langle\rangle, \langle a\rangle, \langle a, ✓\rangle\}$. Hence, it is straightforward that pserials(t) = *traces*(trans(t)) $\setminus$ ✓.

Induction step:

Case $t = OR(t_1, \ldots, t_n)$**:** It is straightforward that
- $[\![t]\!]_{\mathcal{SP}} = \bigcup_{i=1,\ldots,n} [\![t_i]\!]_{\mathcal{SP}}$, and
- *traces*(trans(t)) = *traces*(trans(t_1) $\Box \ldots \Box$ trans(t_i)).

Then, we have

$$\begin{aligned}
tr \in \text{pserials}(t) &\Leftrightarrow \exists\, G \in [\![t]\!]_{\mathcal{SP}} : tr \in \text{pserials}(G)\\
&\Leftrightarrow \exists\, i \in \{1,\ldots,n\}, G \in [\![t_i]\!]_{\mathcal{SP}} : tr \in \text{pserials}(G)\\
&\Leftrightarrow \exists\, i \in \{1,\ldots,n\} : tr \in \text{pserials}(t_i)\\
&\Leftrightarrow tr \in \mathit{traces}(\text{trans}(t_i)) \setminus ✓ \text{ by induction hypothesis}\\
&\Leftrightarrow tr \in \mathit{traces}(\text{trans}(t)) \setminus ✓.
\end{aligned}$$

Case $t = AND(t_1, \ldots, t_n)$**:** It is obvious that:
- $[\![t]\!]_{\mathcal{SP}} = \{G_1 \parallel \ldots \parallel G_n \mid G_i \in [\![t_i]\!]_{\mathcal{SP}}\, \forall\, i = 1, \ldots, n\}$, and
- *traces*(trans(t)) = *traces*(trans(t_1) $\mid\mid\mid \ldots \mid\mid\mid$ trans(t_i)).

Then, we have

$$\begin{aligned}
tr \in \text{pserials}(t) &\Leftrightarrow \exists\, G \in [\![t]\!]_{\mathcal{SP}} : tr \in \text{pserials}(G)\\
&\Leftrightarrow \forall\, i \in \{1,\ldots,n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}} :\\
&\qquad tr \in \text{pserials}(G_1 \parallel \ldots \parallel G_n)\\
&\Leftrightarrow \forall\, i \in \{1,\ldots,n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}}, tr' \in \text{serials}(G_1 \parallel \ldots \parallel G_n) :\\
&\qquad tr \leq tr'\\
&\Leftrightarrow \forall\, i \in \{1,\ldots,n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}}, tr_i \in \text{serials}(G_i) :\\
&\qquad tr' \in tr_1 \mid\mid\mid \ldots \mid\mid\mid t_n \wedge tr \leq tr'\\
&\Leftrightarrow \forall\, i \in \{1,\ldots,n\}, \exists\, tr_i \in \mathit{traces}(\text{trans}(t_i)) \setminus ✓ :\\
&\qquad tr' \in tr_1 \mid\mid\mid \ldots \mid\mid\mid t_n \wedge tr \leq tr' \text{ by induction hypothesis}\\
&\Leftrightarrow \forall\, i \in \{1,\ldots,n\}, \exists\, tr'_i \leq tr_i \in \mathit{traces}(\text{trans}(t_i)) \setminus ✓ :\\
&\qquad tr \in tr'_1 \mid\mid\mid \ldots \mid\mid\mid t'_n\\
&\Leftrightarrow tr \in \mathit{traces}(\text{trans}(t)) \setminus ✓.
\end{aligned}$$

Case $t = SAND(t_1, \ldots, t_n)$**:** It is obvious that:
- $[\![t]\!]_{\mathcal{SP}} = \{ G_1 \cdot \ldots \cdot G_n \mid G_i \in [\![t_i]\!]_{\mathcal{SP}} \forall\ i = 1, \ldots, n \}$, and
- $\mathit{traces}(\text{trans}(t)) = \mathit{traces}(\text{trans}(t_1);\ \ldots;\ \text{trans}(t_n))$.

Then we have:

$$
\begin{aligned}
tr \in \text{pserials}(t) &\Leftrightarrow \exists\, G \in [\![t]\!]_{\mathcal{SP}} : tr \in \text{pserials}(G) \\
&\Leftrightarrow \forall\, i \in \{1, \ldots, n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}} : \\
&\qquad tr \in \text{pserials}(G_1 \cdot \ldots \cdot G_n) \\
&\Leftrightarrow \forall\, i \in \{1, \ldots, n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}}, tr' \in \text{serials}(G_1 \cdot \ldots \cdot G_n) : \\
&\qquad tr \leq tr' \\
&\Leftrightarrow \forall\, i \in \{1, \ldots, n\}, \exists\, G_i \in [\![t_i]\!]_{\mathcal{SP}}, tr_i \in \text{serials}(G_i) : \\
&\qquad tr' \in tr_1 \frown \ldots \frown t_n \wedge tr \leq tr' \\
&\Leftrightarrow \forall\, i \in \{1, \ldots, n\}, \exists\, tr_i \in \mathit{traces}(\text{trans}(t_i)) \setminus \checkmark : \\
&\qquad tr' \in tr_1 \frown \ldots \frown t_n \wedge tr \leq tr' \text{ by induction hypothesis} \\
&\Leftrightarrow \forall\, i \in \{1, \ldots, n\}, \exists\, tr'_i \leq tr_i \in \mathit{traces}(\text{trans}(t_i)) \setminus \checkmark : \\
&\qquad tr \in tr'_1 \frown \ldots \frown t'_n \\
&\Leftrightarrow tr \in \mathit{traces}(\text{trans}(t)) \setminus \checkmark .
\end{aligned}
$$

5 Implementation

We provide a prototype implementation of our proposed methodology. This implementation is built using Python 2.7 and requires as input an attack tree and a formal model of the SUT. It then automatically carries out three main tasks: (1) translates an input attack tree into a CSP process; (2) uses this process and the formal model of the SUT to generate test cases; and (3) executes all the generated test cases by associating each one with a sequence of predefined primitive test scripts. Task 1 is a straightforward implementation of function `trans` from Section 4.1. In the rest of this section, we discuss the implementation of tasks 2 and 3 in detail.

5.1 Test Case Generation

Let us assume that the formal model of the SUT is given as a CSP process *Sys*. Furthermore, the behaviours of the attacker are also given in terms of an attack tree t, which is then transformed into a CSP process $\text{trans}(t)$. We shall use trace refinement in CSP to extract test cases following [16]. To this end, $\text{trans}(t)$ acts as a filter criterion to select test cases among all possible runs of the system captured by *Sys*. As in [16], we define a fresh event *attackSucceed* to mark the end of an attack, which indicates that an attack is successfully executed. We form the following filter

$$\mathit{TestPurpose} = \text{trans}(t);\ (\mathit{attackSucceed} \rightarrow \mathit{Stop})$$

which captures all attacks extended with the marking event *attackSucceed* at the end. Then, we establish the following trace refinement:

$$\mathit{Sys} \,\square\, \mathit{TestCases} \sqsubseteq_T \mathit{Sys} \underset{\Sigma \setminus \{\mathit{attackSucceed}\}}{\parallel} \mathit{TestPurpose}$$

In this refinement, *TestCases* encodes test cases that have previosuly been generated. By combining it with *Sys* using the external choice operator, a fresh test case, i.e., different from the generated ones, will be generated if one exists. $Sys \underset{\Sigma \setminus \{attackSucceed\}}{\parallel} TestPurpose$ encapsulates all attack traces that can be carried out with respect to the formal model *Sys*. These attack traces are ended with the marking event *attackSucceed*, which does not belong to *Sys*, hence, gives rise to counter examples of the refinement. Initially, $TestCases = TestCases_0 = Stop$, i.e., corresponding to an empty set of test cases. This refinement is checked by calling FDR [26]. If an attack trace exists, FDR will provides a counter example of the form $\langle a_1, \ldots, a_n, attackSucceed \rangle$ where $a_1, \ldots, a_n \in \Sigma \setminus \{attackSucceed\}$. We encode this trace as a test case $tc_1 = a_1 \rightarrow \ldots \rightarrow a_n \rightarrow attackSucceed \rightarrow Stop$. After *TestCases* is rebuilt as $TestCases = TestCases_1 = TestCases_0 \,\Box\, tc_1$, the above refinement check is called again and again to extract further test cases $tc_2, \ldots$ and to construct $TestCase_2, \ldots$ until no further counter example can be found. In this implementation, the calls to checking refinements and extracting counter examples are facilitated by API functions provided by FDR [26].

5.2 Test Case Execution

Test cases that are generated can now be assigned programmatic functions that would allow for execution. This is dependant on implementation of the system and so would necessarily be specific rather than abstract. Furthermore, as the attack tree in this case is based on penetration testing, not all actions (such as "social engineering") are scriptable, largely due to requiring manual intervention. All such actions are indicated in the implementation.

Given an attack tree t, let $scriptable(t)$ denote the set of its scriptable leaves. Then, each scriptable leaf $a \in scriptable(t)$ is associated with a primitive test script $script(a)$. A generated test case $tc = a_1 \rightarrow \ldots \rightarrow a_n \rightarrow attackSucceed \rightarrow Stop$ is automatically executable if $a_i \in scriptable(t)$ for all $i = 1, \ldots, n$. Then, executing an automatically executable test case means to execute all test scripts $script(a_1), \ldots, script(a_n)$ sequentially. If all such scripts are executed successfully, the test case is called *passed*, otherwise *failed*. Note that a passed test case means that the SUT is not secure with respect to the attack encoded by this test case. Conversely, the SUT is impervious to this attack.

In this paper, we use the example of an aftermarket on-board diagnostics (OBD-II) dongle attached to the vehicle (see Section 5). Executable test cases are written in Python 2.7 to enable compatibility with Bluetooth functions.

6 Case study

We take here a case study of evaluating the intra-vehicular network with the attack goal of vehicle compromise. The attack tree (see Section 6.2) for this goal is based around access through a Bluetooth-enabled aftermarket device that attaches to the vehicle's on-board diagnostic (OBD-II) port. These devices were

originally created so that enthusiasts and hobbyists were able to read information from their own vehicles for diagnostic and maintenance purposes. The devices (or dongles) contain an ELM327 chip [7] which serves as an RS-232 interpreter. "Attention Modem" (`AT`) commands are used to configure the chip through any serial terminal. The device used for this case study was the OBDLINK MX, with an ELM chip version of 1.3, attached to a 2013 small hatchback from a major manufacturer. Through such a device, an attacker is able to gain access to the intra-vehicular CAN bus remotely and potentially push messages directly into the vehicle [1, 2]. The risk of compromise is exacerbated by the fact that these OBD-II devices are usually highly insecure, being wireless, and with weak PINs that are made public (such as `0000` or `1234`) [17].

6.1 Vehicular Communications

Messages that are sent through the dongle on to the CAN bus takes the form of either a raw Controller Area Network (CAN) frame (Section 6.1) or a diagnostic message (Section 6.1) (that is translated into a CAN frame by the dongle).

CAN Messages. The CAN protocol is the primary mode of communication inside the vehicle. The latest version is CAN 2.0, first specified in 1991 [19] and embodied as an ISO standard (ISO11898) in 2003.

The standard CAN packet comprises (up to) 11 bits for the message ID, followed by (up to) 8 bytes of data, then a cyclic redundancy check (16 bits) for error detection. The full 8 bytes of data need not be used. Information for a door sensor, for example, may only require 1 bit. Conversely a message can be spread across many frames.

Arbitration, should nodes on the CAN network transmit simultaneously, is based on message prioritisation. This prioritisation is determined using the message ID, with the lowest ID being the highest priority; implementation usually means that mission-critical messages are the ones assigned lower IDs.

Assignment of IDs along with data payload is manufacturer specific, however, reuse is common to save on the cost of redesigning a network [18].

Reverse engineering of CAN messages is difficult considering volume and variety of content that is transmitted. This is especially the case without an Original Equipment Manufacturer's (OEM's) typically confidential CAN database, which contains definitions for every message and signal. However, specific CAN messages for discrete events (such as unlocking doors) can be obtained relatively easily through trial and error.

CAN data is transmitted in a bus configuration; any Electronic Control Unit (ECU) on the network has access to all messages. There is no addressing; each ECU listens to a set of IDs which then triggers pre-determined functionality.

Diagnostic Messages. Parameter IDs (PIDs) are used to perform diagnostic functions or request data from the vehicle specifically through OBD-II port; done through a query-response mechanism where a PID query comprises the CAN ID `7DF` followed by 8 data bytes. The first byte is data length (usually 02) with the second byte the *mode* and the third byte typically the PID. The combination of modes and PIDs can then be transmitted and a response received from whatever

in-vehicle module is responsible. The response CAN ID is typically 8 (in hex) higher than the message ID that the responding ECU answers to.

The first ten modes (`01` to `0A`, described in SAE J1979 (E/E Diagnostic Test Modes) [22], are standard to all compliant vehicles: PID is only the 2nd byte, with the 3rd to 8th byte unused. With non-standard modes, PIDs could extend to 3rd byte. Manufacturers, are not obliged to implement all standard commands, and additionally could also define functions for non-standard PIDs. There is much information that could be gathered through this port. For example, sending the mode `09` with PID `02` retrieves the Vehicle Identification Number (VIN), which is unique to vehicles, used for maintenance to recovery of stolen vehicles.

6.2 Attack Tree Translation

The attack tree used for this case study is shown in Figure 2. Figure 2 also lists test scripts corresponding to leaves in this attack tree. If a leaf is not scriptable, it is denoted as a *manual leaf*.

The function trans(Vehicle Compromise) (see Section 4.1), gives the translation of this tree into CSP as below:

$$
\begin{aligned}
&\mathit{Attacker} = \mathit{Vehicle\_Compromise}\\
&\mathit{Vehicle\_Compromise} = \mathit{Connect\_to\_device};\ \mathit{Cause\_Vehicle\_Compromise}\\
&\mathit{Connect\_to\_device} = \mathit{Using\_legitimate\_device} \,\Box\, \mathit{Spoof\_previously\_paired\_device}\\
&\mathit{Using\_legitimate\_device} = \mathit{Determine\_pairing\_status};\ \mathit{Connect\_to\_serial\_port}\\
&\mathit{Determine\_pairing\_status} = \mathit{action\_Determine\_pairing\_status} \rightarrow \mathit{Skip}\\
&\mathit{Connect\_to\_serial\_port} = \mathit{action\_Connect\_to\_serial\_port} \rightarrow \mathit{Skip}\\
&\mathit{Spoof\_previously\_paired\_device} =\\
&\qquad \mathit{Find\_the\_link\_key\_from\_local\_or\_remote\_device}\\
&\quad ||| \ \mathit{Change\_address\_of\_local\_device}\\
&\mathit{Change\_address\_of\_local\_device} =\\
&\quad \mathit{action\_Change\_address\_of\_local\_device} \rightarrow \mathit{Skip}\\
&\mathit{Find\_the\_link\_key\_from\_local\_or\_remote\_device} =\\
&\quad \mathit{action\_Find\_the\_link\_key\_from\_local\_or\_remote\_device} \rightarrow \mathit{Skip}\\
&\mathit{Cause\_Vehicle\_Compromise} =\\
&\qquad \mathit{Using\_OBD\_messages}\\
&\quad \Box\ \mathit{Run\_through\_all\_messages}\\
&\quad \Box\ \mathit{Flooding\_with\_raw\_CAN\_messages}\\
&\mathit{Flooding\_with\_raw\_CAN\_messages} =\\
&\quad \mathit{Predetermine\_CAN\_messages};\ \mathit{Send\_\_flood\_with\_\_CAN\_messages}\\
&\mathit{Predetermine\_CAN\_messages} =\\
&\qquad \mathit{Using\_passive\_monitoring}\\
&\quad \Box\ \mathit{Using\_OEM\_CAN\_database}\\
&\quad \Box\ \mathit{Using\_reverse\_engineering}\\
&\mathit{Send\_flood\_with\_CAN}_{m}\mathit{essages} = \mathit{action\_Send\_flood\_with\_CAN\_messages} \rightarrow \mathit{Skip}\\
&\mathit{Using\_OEM\_CAN\_database} = \mathit{action\_Using\_OEM\_CAN\_database} \rightarrow \mathit{Skip}\\
&\mathit{Using\_passive\_monitoring} = \mathit{action\_Using\_passive\_monitoring} \rightarrow \mathit{Skip}\\
&\mathit{Using\_reverse\_engineering} = \mathit{action\_Using\_reverse\_engineering} \rightarrow \mathit{Skip}
\end{aligned}
$$

$Run\_through\_all\_messages = Run\_through\_standard;\ Run\_through\_non\_standard$
$Run\_through\_standard = action\_Run\_through\_standard \rightarrow Skip$
$Run\_through\_non\_standard = action\_Run\_through\_non\_standard \rightarrow Skip$
$Using\_OBD\_messages = Flood\_with\_set\_OBD\_messages$
$Flood\_with\_set\_OBD\_messages = action\_Flood\_with\_set\_OBD\_messages \rightarrow Skip$

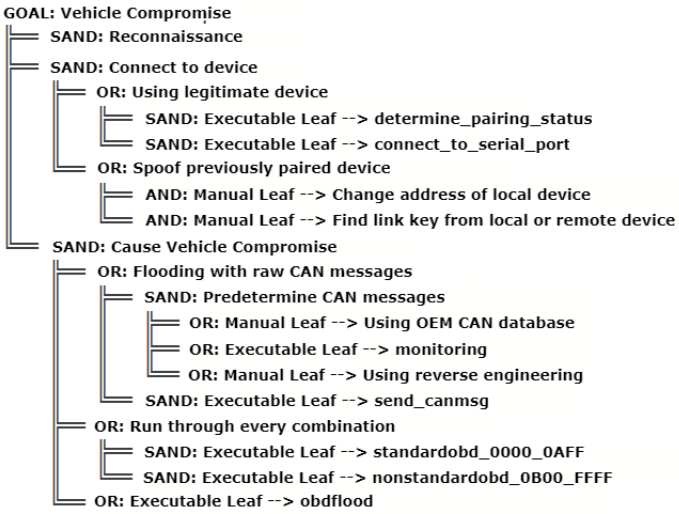

Fig. 2. Attack tree, with attack goal of compromising the vehicle through an aftermarket Bluetooth-enabled OBD-II device

Reconnaissance was defined as "to find as much information as possible" (meaning the subsequently generated formal attack tree would be much larger). Many of the steps were manual and non-sequential.

6.3 Results

We generate test cases using the implementation in Section 5.1. Given the small size of the attack tree, we use the most abstract model for the SUT where all behaviours are accepted (the most insecure model) to generate a total of 15 test cases. Results from the run of test cases against an actual implementation are given in Figure 1. Three of the test cases passed (i.e. they were executed successfully) (see Figure 3). We highlight test case 3 ($TC3$ in Figure 3) for further analysis.

The action of flooding with a particular diagnostic message resulted in loss of function in the vehicle of both electronics and engine. This violates the security property of availability by causing a denial of service. Additionally, injection of messages into the CAN bus also changes the stream of CAN bus signals that would normally be expected in vehicles. This violates the security property of

TC#	Execution result
1	unexecutable action_Find_the_link_key_from_local_or_remote_device
2	unexecutable action_Change_address_of_local_device
3	Passed
4	unexecutable action_Find_the_link_key_from_local_or_remote_device
5	unexecutable action_Change_address_of_local_device
6	Passed
7	unexecutable action_Find_the_link_key_from_local_or_remote_device
8	Passed
9	unexecutable action_Using_OEM_CAN_database
10	unexecutable action action_Using_reverse_engineering
11	unexecutable action_Change_address_of_local_device
12	unexecutable action_Find_the_link_key_from_local_or_remote_device
13	unexecutable action_Change_address_of_local_device
14	unexecutable action_Change_address_of_local_device
15	unexecutable action_Find_the_link_key_from_local_or_remote_device

Table 1. Test cases that were run against a real world vehicle

$TC(3) = action\_Determine\_pairing\_status \rightarrow$
$action\_Connect\_to\_serial\_port \rightarrow action\_Flood\_with\_set\_OBD\_messages \rightarrow$
$attack\_succeed \rightarrow Stop$
$TC(6) = action\_Determine\_pairing\_status \rightarrow$
$action\_Connect\_to\_serial\_port \rightarrow action\_Run\_through\_standard \rightarrow$
$action\_Run\_through\_non\_standard \rightarrow attack\_succeed \rightarrow Stop$
$TC(8) = action\_Determine\_pairing\_status \rightarrow$
$action\_Connect\_to\_serial\_port \rightarrow action\_Using\_passive\_monitoring \rightarrow$
$action\_Send\_flood\_with\_\_CAN\_messages \rightarrow attack\_succeed \rightarrow Stop$

Fig. 3. Test cases that succeeded

integrity (in which no unauthorised modification should be allowed). Protecting against this could involve the addition of gateways in the SUT, which could either filter out floods of messages (by defining thresholds for the number of these messages that could be sent through at any given time). Alternatively, such messages (unless from an authorised source) could be disallowed completely.

The other test cases (all involving permutations of finding a link key, and changing the address) were not scripted because they required manual intervention. The former because it would need a remote device set to enable logging on the Host Controller Interface (not always possible) or to manually acquire data from a vehicle to find where the link key has been stored (which would have required hardware removal). The latter is automatable (for example, using a tool called *Spooftooph* [6]), however, either hardware removal or social manipulation is involved to find knowledge of an address that is already stored on the vehicle.

Other branches that were unscripted involves reverse engineering CAN messages to inject, which involves manual trial and error due to the sheer volume and variety of messages that are on the CAN bus at any one time. Using an OEM

CAN database would enable automation, but availability is often non-existent due to commercial confidentiality. The branch that ended with successful test cases all involved using a legitimate device. That is, a device that was under our control, which we could use to test weaknesses in the vehicular implementation.

7 Conclusion and Future Work

We have demonstrated the translation of an informal attack tree into a formal structure using the process algebra CSP and proved equivalence. We use this tree to generate test cases automatically, and assign executions to scriptable test cases. We execute the test cases on a real-world vehicle (although this could be substituted with a testbed, with input from an OEM to reflect a real architecture, without the cost or risk to a test vehicle [9]). Thus, the full testing process is one step further to automation, and furthermore, the formal model of the attack tree could also be used for formal verification should the specifications of the system-under-test be available. Limitations are around how a tree is created (still largely manual) and certain actions within the attack tree requiring manual intervention. The aim is for the entire process (at an abstract level) to resemble Figure 1. The work with testbeds (as in work done by [9]) as well as manual testing could continue to assist in further refinement of the models created.

References

1. Argus Cybersecurity: Argus Cyber Security Working With Bosch to Promote Public Safety and Mitigate Car Hacking (2017), `http://bit.ly/2tNBLsm`
2. Cheah, M., Bryans, J., Fowler, D.S., Shaikh, S.A.: Threat Intelligence for Bluetooth-enabled Systems with Automotive Applications : An Empirical Study. In: Proceedings of the 47th IEEE/IFIP Dependable Systems and Networks Workshops: Security and Safety in Vehicles (SSIV). IEEE, Denver (Jun 2017)
3. Cheah, Madeline and Shaikh, Siraj and Haas, Olivier and Ruddle, Alastair: Towards a systematic security evaluation of the automotive Bluetooth interface. Journ.of Veh. Comms. 9(July), 8–18
4. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T.: Comprehensive Experimental Analyses of Automotive Attack Surfaces. In: Proceedings of 20th USENIX Security Symposium. pp. 77–92. USENIX Association, San Francisco, CA (Aug 2011)
5. Cho, K.T., Shin, K.G.: Error Handling of In-vehicle Networks Makes Them Vulnerable. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security. pp. 1044–1055. ACM New York, NY, Vienna (Oct 2016)
6. Dunning, J.: Spooftooph (2012), `http://bit.ly/2tiOx5O`
7. ELM Electronics: ELM Electronics: OBD, `http://bit.ly/2s0yZPZ`
8. Felderer, M., Zech, P., Breu, R., Buchler, M., Pretschner, A.: Model-based security testing: a taxonomy and systematic classification. Software Testing Verification and Reliability 26(2), 119–148 (2015)
9. Fowler, D.S., Cheah, M., Shaikh, S.A., Bryans, J.: Towards a testbed for automotive cyberecurity. In: Process of the 10th Int. Conf. on Software Testing, Verification and Validation: Industry Track. IEEE, Tokyo, Japan (Mar 2017)

10. Greenberg, A.: Hackers Remotely Kill a Jeep on the HighwayWith Me in It (2015), http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
11. Hoare, C.: Communicating Sequential Processes. Prentice Hall International, UK, electronic edn. (1985)
12. Hoppe, T., Kiltz, S., Dittmann, J.: Security threats to automotive CAN networks - Practical examples and selected short-term countermeasures. Reliability Engineering & System Safety 96(1), 11–25 (Jan 2011)
13. Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., Trujillo-Rasua, R.: Attack Trees with Sequential Conjunction. In: Proceedings of the 30th IFIP TC 11 International Conference. vol. 455, pp. 339–353. Hamburg, Germany (May 2015)
14. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental Security Analysis of A Modern Automobile. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 447–462. IEEE, Oakland, CA (May 2010)
15. Mauw, S., Oostdijk, M.: Foundations of Attack Trees. In: Won, D.H., Kim, S. (eds.) Proceedings of the 8th Int. Conf. on Information Security and Cryptology. pp. 186–198. No. im, Springer Berlin Heidelberg, Seoul, South Korea (2005)
16. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Asp. Comput. 26(3), 441–490 (2014)
17. Oka, D.K., Furue, T., Langenhop, L., Nishimura, T.: Survey of Vehicle IoT Bluetooth Devices. In: 2014 IEEE 7th International Conference on Service-Oriented Computing and Applications. pp. 260–264. IEEE, Matsue, Japan (Nov 2014)
18. Pretschner, A., Broy, M., Kruger, I.H., Stauner, T.: Software engineering for automotive systems: A roadmap. In: Proceedings of the FOSE'07 2007 Future of Software Engineering. pp. 55–71. IEEE, Minneapolis, MN (May 2007)
19. Robert Bosch GmbH: CAN Specification version 2.0 (1991), http://esd.cs.ucr.edu/webres/can20.pdf
20. Roscoe, A.: Understanding Concurrent Systems. Springer, London, 1 edn. (2010)
21. Ruddle, A., Ward, D., Weyl, B., Idrees, S., Roudier, Y., Friedewald, M., Leimbach, T., Fuchs, A., Gurgens, S., Henniger, O., Rieke, R., Ritsscher, M., Broberg, H., Apvrille, L., Pacalet, R., Pedroza, G.: EVITA Project: Deliverable D2.3 - Security requirements for automotive on-board networks based on dark-side scenarios. Tech. rep. (2009), http://www.evita-project.org/Deliverables/EVITAD2.3.pdf
22. SAE International: SAE J1979 E/E Diagnostic Test Modes (2014), http://standards.sae.org/j1979_201408/
23. SAE International: J3061 : Cybersecurity Guidebook for Cyber-Physical Vehicle Systems (2016), http://standards.sae.org/j3061_201601/
24. Salfer, M., Schweppe, H., Eckert, C.: Efficient Attack Forest Construction for Automotive On-board Networks. In: Chow, S.S., Camenisch, J., Hui, L.C., Yiu, S.M. (eds.) 17th International Conference (ISC) on Information Security (Oct 2014)
25. Schneier, B.: Attack Trees: Modeling Security Threats (1999), http://www.schneier.com/paper-attacktrees-ddj-ft.html
26. University of Oxford: FDR3 - The CSP Refinement Checker, https://www.cs.ox.ac.uk/projects/fdr/
27. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test., Verif. Reliab. 22(5), 297–312 (2012)
28. Vigo, R., Nielson, F., Nielson, H.R.: Automated generation of attack trees. In: Proc. of the 27th Computer Security Foundations Symposium. IEEE, Vienna (2014)