SPTool – equivalence checker for SAND attack trees

Barbara Kordy[1,2], Piotr Kordy[3,4], and Yoann van den Boom[5,6]

[1]INSA Rennes, Rennes, France
[2]IRISA, Rennes France
barbara.kordy@irisa.fr
[3]University of Birmingham, Birmingham UK,
[4]University of Luxembourg, Luxembourg, Luxembourg
[5]University Rennes 1, Rennes, France
[6]INRIA, Rennes, France

Abstract. A SAND attack tree is a graphical model decomposing an attack scenario into basic actions to be executed by the attacker. SAND attack trees extend classical attack trees by including the sequential conjunctive operator (SAND) to the formalism. They thus allow to differentiate actions that need to be executed sequentially from those that can be performed in parallel. Since several structurally different SAND attack trees can represent the same attack scenario, it is important to be able to decide which SAND attack trees are equivalent.
SPTool is free, open source software for checking equivalence of SAND attack trees and computing their canonical forms. It relies on term rewriting techniques and an equational theory axiomatizing SAND attack trees.

Keywords: Attack trees, sequential operator, equivalence, rewriting, axiomatization, canonical form, SAND, SPTool, Maude.

1 Introduction and motivation

Attack trees [9] are graphical models aiming to represent and evaluate security vulnerabilities of systems or organizations. An attack tree is a labeled AND-OR tree whose root depicts the ultimate goal of the attacker and the remaining nodes decompose this goal into sub-goals, using disjunctive (OR) and conjunctive (AND) refinements. The leaves of an attack tree represent basic attack steps, i.e., actions that the attacker needs to perform in order to reach his goal. Since attack trees do not only support the representation of security problems, but also help in performing their quantitative analysis [8], the formalism is frequently used in industry as a means to facilitate the risk assessment process [7].

Unfortunately, the simple AND-OR structure is not sufficiently rich to capture all real-life features. One of the main drawbacks of classical attack trees is that they do not distinguish between actions that can be performed in parallel from the ones that need to be executed in a specific order. To overcome this limitation, the authors of [5] have formalized SAND *attack trees* which extend classical attack trees with the sequential conjunctive refinement (SAND) in order to allow the modeling of sequences of actions. `SAND` attack tree depicted in Fig. 1 and described in Example 1 illustrates how to steal money from a bank account.

Example 1. In order to steal money using an ATM machine, the attacker must first get relevant credentials and then withdraw money from the victim's bank account. The root of the tree from Fig. 1 is thus refined using the sequential conjunctive refinement `SAND` (arc with arrow). To get the necessary credentials, the attacker must steal the victim's card and get the corresponding PIN. The order in which the card and the PIN will be obtained is not relevant, thus the standard conjunctive refinement `AND` (simple arc) has been used to refine the 'get credentials' node. In order to get the PIN, the attacker has two options: he can either social engineer the victim to convince her to reveal the secret four digits or find a post-it with the PIN written on it. Since each of these options is sufficient to learn the PIN, the 'get PIN' node has been refined using the disjunctive refinement `OR` (no arc).

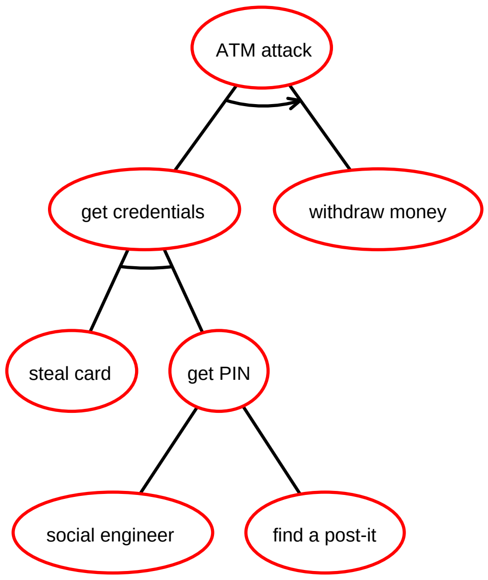


Fig. 1: `SAND` attack tree for an ATM attack

It is well-known that several structurally different (`SAND`) attack trees may be equivalent, i.e., represent the same scenario [9,6,5]. For instance, the ATM attack described in Example 1 can also be illustrated using the tree from Fig. 2. Both representations are useful and have their strengths. The tree from Fig. 1 is more concise and represents how the attacker will mount his attack. The one from Fig. 2 is in *canonical form* – it enumerates all possible attack vectors explicitly. It thus represents possible executions of the attack.

In order to decide which (`SAND`) attack trees are equivalent, numerous formal semantics (based on Boolean functions [8], multisets [9,6], series-parallel graphs [5]) have been introduced. The choice of an appropriate semantics is closely related to the type of quantitative analysis to be performed. For instance, if we want to evaluate probability of an attack, we can use the Boolean interpretation of attack trees, as in [8]. However, if the attack time or cost are being evaluated, the multiset-based semantics needs to be used [9]. The relation between formal semantics for the attack tree-like models and their quantitative analysis has been formalized in [6]. It relies on the notion of compatibility which

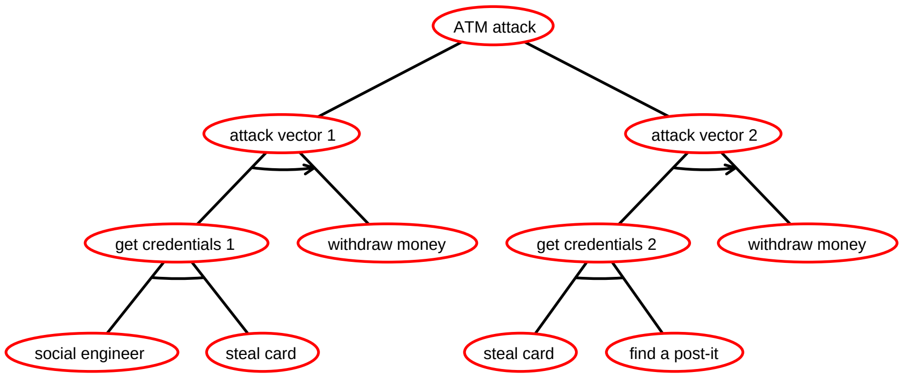


Fig. 2: ATM attack in canonical form

guarantees that the quantification on equivalent trees yields the same value. It is thus important to be able to quickly verify whether two trees are equivalent.

In addition, attack trees used in practice grow quite fast and may reach several thousand of nodes. For instance, in the Galileo risk assessment program attack trees stretching over 40 A4 pages have been considered [10]. Manual handling of such trees becomes infeasible and needs therefore to be automated. However, none of existing tools supporting drawing, automated generation, and quantitative analysis of attack tree-like models, such as [3,11,4,1], can handle semantics preserving transformations of attack trees with sequential conjunction and compare structurally different trees.

To support the use of SAND attack trees, we have implemented a prototype tool, called SPTool, integrating graphical security modeling and term rewriting. SPTool provides the canonical form for SAND attack trees and checks whether two SAND attack trees are equivalent. Since SAND attack trees form a conservative extension of attack trees, as formalized in [9], SPTool can also be employed to reason about classical attack trees which use OR and AND refinements only.

Section 2 gives an overview of the formal foundations for SAND attack trees which are relevant for the understanding of SPTool. The main features of SPTool, its architecture, and implementation characteristics are described in Section 3. We evaluate the performance of SPTool in Section 4 and conclude in Section 5.

2 SAND attack trees formally

SAND attack trees are closed terms over the signature $\mathbb{B} \cup \{\mathtt{OR}, \mathtt{AND}, \mathtt{SAND}\}$, generated by the following grammar, where OR, AND, and SAND are unranked operators, $\mathbb{B}$ is the set of terminal symbols, and $b \in \mathbb{B}$:

$$t ::= b \mid \mathtt{OR}(t, \dots, t) \mid \mathtt{AND}(t, \dots, t) \mid \mathtt{SAND}(t, \dots, t).$$

Fig. 3 displays the set of equations, introduced in [5] and denoted by $E_{\mathcal{SP}}$, which axiomatize `SAND` attack trees. These equations express the properties of the refining operators `OR`, `AND`, and `SAND` and thus encode the transformations that do not change the meaning of a `SAND` attack tree.

$$\begin{aligned}
&\mathtt{OR}(Y_1,\dots,Y_\ell) = \mathtt{OR}(Y_{\sigma(1)},\dots,Y_{\sigma(\ell)}), \quad \forall\sigma\in\mathrm{Sym}_\ell && (E_1)\\
&\mathtt{AND}(Y_1,\dots,Y_\ell) = \mathtt{AND}(Y_{\sigma(1)},\dots,Y_{\sigma(\ell)}), \quad \forall\sigma\in\mathrm{Sym}_\ell && (E_2)\\
&\mathtt{OR}\big(\overline{X},\mathtt{OR}(\overline{Y})\big) = \mathtt{OR}(\overline{X},\overline{Y}) && (E_3)\\
&\mathtt{AND}\big(\overline{X},\mathtt{AND}(\overline{Y})\big) = \mathtt{AND}(\overline{X},\overline{Y}) && (E_4)\\
&\mathtt{SAND}\big(\overline{X},\mathtt{SAND}(\overline{Y}),\overline{Z}\big) = \mathtt{SAND}(\overline{X},\overline{Y},\overline{Z}) && (E_{4'})\\
&\mathtt{OR}(A) = A && (E_5)\\
&\mathtt{AND}(A) = A && (E_6)\\
&\mathtt{SAND}(A) = A && (E_{6'})\\
&\mathtt{AND}\big(\overline{X},\mathtt{OR}(\overline{Y})\big) = \mathtt{OR}\big(\mathtt{AND}(\overline{X},Y_1),\dots,\mathtt{AND}(\overline{X},Y_\ell)\big) && (E_{10})\\
&\mathtt{SAND}\big(\overline{X},\mathtt{OR}(\overline{Y}),\overline{Z}\big) = \mathtt{OR}\big(\mathtt{SAND}(\overline{X},Y_1,\overline{Z}),\dots,\mathtt{SAND}(\overline{X},Y_\ell,\overline{Z})\big) && (E_{10'})\\
&\mathtt{OR}(A,A,\overline{X}) = \mathtt{OR}(A,\overline{X}). && (E_{11})
\end{aligned}$$

Fig. 3: The set $E_{\mathcal{SP}}$ of equations axiomatizing `SAND` attack trees, where $k, m \geq 0$, $\ell \geq 1$, $\overline{X} = X_1,\dots,X_k$, $\overline{Y} = Y_1,\dots,Y_\ell$, and $\overline{Z} = Z_1,\dots,Z_m$ are sequences of variables and A is a variable. Sym_ℓ denotes the set of all bijections from $\{1,\dots,\ell\}$ to itself. The numbering of axioms is explained in [5].

Definition 1. *We say that two* `SAND` *attack trees are* equivalent *if they can be obtained from each other by applying a finite number of axioms from $E_{\mathcal{SP}}$.*

In other words, `SAND` attack trees are equivalent if they are equal modulo the axioms from $E_{\mathcal{SP}}$. For instance, we can easily check by applying axioms (E_{10}), $(E_{10'})$, and (E_2) that the `SAND` attack trees from Fig. 1 and Fig. 2 are equivalent.

The authors of [5] have proven that by orienting axioms (E_3), (E_4), $(E_{4'})$, (E_5), (E_6), $(E_{6'})$, (E_{10}), $(E_{10'})$, and (E_{11}) from left to right, one obtains a terminating and confluent term rewriting system, which we denote by $R_{\mathcal{SP}}$. Using $R_{\mathcal{SP}}$, every `SAND` attack tree can be transformed into an equivalent one in *canonical form*, i.e., which is in normal form wrt $R_{\mathcal{SP}}$. For instance, the tree from Fig. 2 is a canonical form of the one from Fig. 1. Due to the confluence and the termination of $R_{\mathcal{SP}}$, we obtain the following result.

Proposition 1. *Two* `SAND` *attack trees are equivalent if and only if their normal forms with respect to $R_{\mathcal{SP}}$ are equal modulo commutativity and associativity of* `OR` *and* `AND` *and modulo associativity of* `SAND`.

Thus, the rewriting system $R_{\mathcal{SP}}$ yields an effective way for handling `SAND` attack trees and provides formal foundations for SPTool described in the next section.

3 SPTool software

SPTool is a free and open source tool allowing to reason about `SAND` attack trees. It offers two main functionalities:

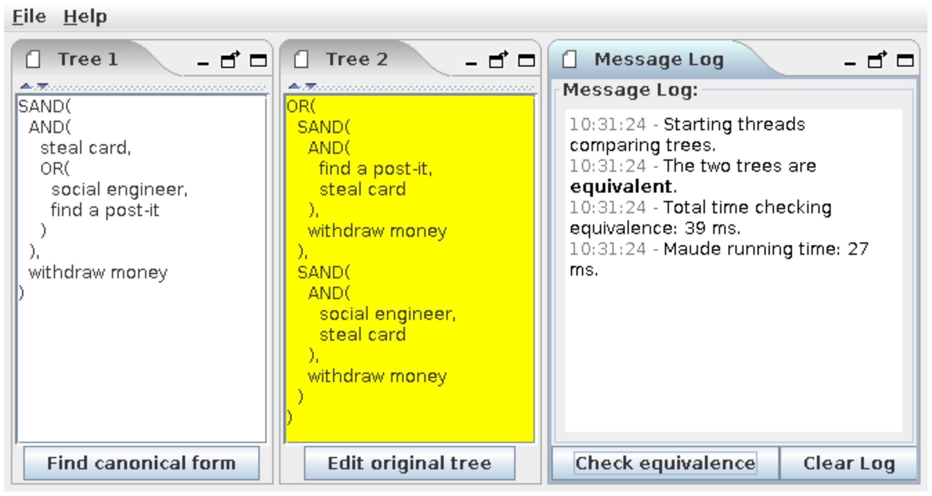
1. Given a `SAND` attack tree t, SPTool returns the canonical form of t;
2. SPTool checks whether two `SAND` attack trees are equivalent wrt $E_{\mathcal{SP}}$.

SPTool relies on the term rewriting system $R_{\mathcal{SP}}$, thus we have decided to interface it with Maude – 'a language and system supporting both equational and rewriting logic specification and programming for a wide range of applications' [2]. The choice of Maude was motivated by the fact that it is able to handle rewriting modulo theories (in our case associativity and commutativity). We can thus work with unranked operators `OR`, `AND`, and `SAND` directly. Maude specification file implementing the system $R_{\mathcal{SP}}$ is illustrated in Fig. 4.

```
*** library & type
protecting STRING .
sorts void adterm term .
subsort void adterm < term .
*** operators
op nil : -> void [ctor] .
op basic : String -> adterm [ctor] .
op OR : adterm adterm -> adterm [ctor assoc comm id: nil] .
op AND : adterm adterm -> adterm [ctor assoc comm id: nil] .
op SAND : adterm adterm -> adterm [ctor assoc id: nil] .
*** variables used
vars x y z : adterm .
*** rewrite rules
eq [e10] : AND(x,OR(y,z)) = OR(AND(x,y),AND(x,z)) .
eq [e101] : SAND(x,OR(y,z)) = OR(SAND(x,y), SAND(x,z)) .
eq [e102] : SAND(OR(y,z),x) = OR(SAND(y,x),SAND(z,x)) .
eq [e11] : OR(x,x) = x .
```

Fig. 4: Maude specification for the term rewriting system $R_{\mathcal{SP}}$

A user interacts with SPTool via the GUI shown in Fig. 5. After having provided input `SAND` attack tree(s) (in windows *Tree 1* and/or *Tree 2*), he selects either to compute the corresponding canonical form (*Find canonical form* button) or to check equivalence between two trees (*Check equivalence* button). In order to find the canonical form of a tree, SPTool uses Maude which is launched in a separate thread for each tree. After having computed the canonical form, the user can switch between seeing the original tree and its canonical form. To distinguish between the two forms easily, the canonical form is displayed on yellow background. The time performance of SPTool is given in the *Message*



```
SAND(
 AND(
  steal card,
  OR(
   social engineer,
   find a post-it
  )
 ),
 withdraw money
)
```

```
OR(
 SAND(
  AND(
   find a post-it,
   steal card
  ),
  withdraw money
 ),
 SAND(
  AND(
   social engineer,
   steal card
  ),
  withdraw money
 )
)
```

```
Message Log:
10:31:24 - Starting threads
comparing trees.
10:31:24 - The two trees are
equivalent.
10:31:24 - Total time checking
equivalence: 39 ms.
10:31:24 - Maude running time: 27
ms.
```

Fig. 5: Graphical user interface of SPTool

log window. While calculating a canonical form, two parameters are displayed: *Maude running time* – the time that Maude required to rewrite the input tree to its normal form; and *Total time* – the time SPTool took to interact with Maude, rewrite the tree, parse it, and display the result. In order to check whether two `SAND` attack trees are equivalent, SPTool first finds their canonical forms and then applies Proposition 1 to draw the conclusion. The answer is displayed in the *Message log* window together with the corresponding time performance.

SPTool offers a possibility of loading `SAND` attack trees (stored as txt files) and saving the results (i.e., trees in canonical form and the content of the *Message log* window). The format of SPTool files is compatible with ADTool – software allowing to display and quantitatively analyze attack tree-like models [3]. On the one hand, the trees drawn with ADTool can be opened using SPTool to compute their canonical forms. On the other hand, the output trees of SPTool can be graphically visualized and quantitatively analyzed using ADTool.

The architecture of SPTool is presented in Fig. 6. The tool is implemented in Java and requires Java SE 7 or later. It uses Docking Frames library as a docking framework. Due to the use of Maude, SPTool runs on Linux platform. Our software is freely available at `http://people.irisa.fr/Barbara.Kordy/sptool`. It is distributed as a jar package.

4 Experimental results

For the purpose of testing our software, we have randomly generated `SAND` attack trees with varying number of nodes. For each of these trees, we have calculated the corresponding canonical form using SPTool. The experiments were performed on a 64-bit Debian Linux machine with Intel i7-4600U processor and 8GB of

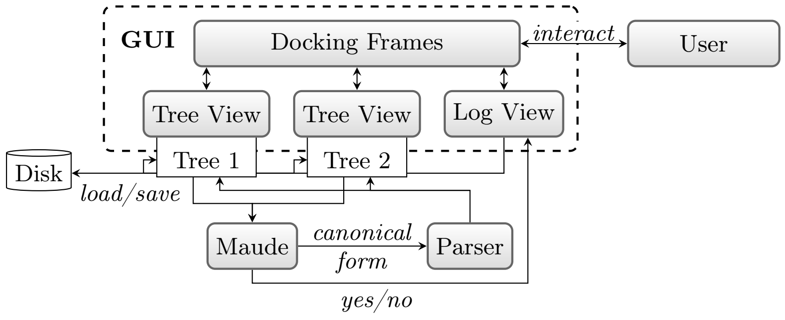


Fig. 6: An overview of the SPTool architecture

memory. Table 1 presents a representative selection of our tests. We notice that the canonical form of a `SAND` attack tree can be exponentially larger than the input tree. This is especially due to the axioms (E_{10}) and ($E_{10'}$) encoding the distributivity property, which multiply parts of a tree, if applied from left to right. Large size of canonical forms implies that the values of the *Maude running time* and the *Total time* parameters differ substantially. Indeed, in the case of large canonical forms, parsing the tree produced by Maude and displaying it as a formatted string on the screen takes significant time. Our experiments have shown that SPTool handles about 150k – 300k nodes per second and that it scales linearly.

Input tree		**Calculation time** (ms)		**Canonical form**	
Non-leaf nodes	Leaves	Maude running time	Total time	Non-leaf nodes	Leaves
73	143	1 206	8 188	264 240	1 709 520
48	94	880	6 921	270 000	1 045 800
39	94	800	2 464	80 191	971 030
42	118	176	529	18 430	149 848
51	104	81	282	6 912	66 816
29	71	28	70	381	1 696
26	66	29	89	218	1 025

Table 1: Calculation of canonical form for randomly generated trees

We have also employed SPTool to compute canonical forms of manually created `SAND` attack trees corresponding to real-life scenarios. We have observed that trees produced by humans admit much smaller canonical forms compared to randomly generated trees of similar size. This can be explained by the fact that manually created trees are more structured and their format is close to the canonical form. In consequence, handling of real-life `SAND` attack trees requires less rewrite steps and is much faster compared to automatically generated trees.

5 Conclusion

`SAND` attack trees are a popular and practical extension of attack trees allowing to distinguish between actions that need to be performed in a predefined order and those that can be executed in parallel. In this paper, we have presented SPTool – prototype software which makes use of term rewriting to transform a `SAND` attack tree into its canonical form and to check equivalence between two `SAND` attack trees. The tool can also be employed to handle classical attack trees which use standard `OR` and `AND` refinements only.

The performance tests executed with SPTool have shown that the rewriting theory provides a practical and efficient method to find canonical forms of attack trees and to check their equivalence. These tests allowed us to validate the concept of axiomatization of attack tree-like models, as developed in [9] and [6].

Our future work will focus on formalization and axiomatization of attack–defense trees with sequential conjunction, a model which augments the expressive power of `SAND` attack trees by explicitly including countermeasure nodes.

Acknowledgments The research leading to these results has received funding from the EU Seventh Framework Programme (TREsPASS, grant number 318003) and from FNR Luxembourg (ADT2P, grant number C13/IS/5809105).

References

1. Amenaza: SecurITree. `http://www.amenaza.com/SS-what_is.php` (2001–2012)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag (2007)
3. Gadyatskaya, O., Jhawar, R., Kordy, P., Lounis, K., Mauw, S., Trujillo-Rasua, R.: Attack Trees for Practical Security Assessment: Ranking of Attack Scenarios with ADTool 2.0. In: QEST 2016. LNCS, vol. 9826, pp. 159–162. Springer (2016)
4. Isograph: AttackTree+. `http://www.isograph.com/software/attacktree/`
5. Jhawar, R., Kordy, B., Mauw, S., Radomirovic, S., Trujillo-Rasua, R.: Attack Trees with Sequential Conjunction. In: IFIP SEC 2015. IFIP AICT, vol. 455, pp. 339–353. Springer (2015)
6. Kordy, B., Mauw, S., Radomirovic, S., Schweitzer, P.: Attack–Defense Trees. J. Log. Comput. 24(1), 55–87 (2014)
7. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Computer Science Review 13-14, 1–38 (2014)
8. Kordy, B., Pouly, M., Schweitzer, P.: Probabilistic reasoning with graphical security models. Information Sciences 342, 111–131 (2016)
9. Mauw, S., Oostdijk, M.: Foundations of Attack Trees. In: ICISC 2005. LNCS, vol. 3935, pp. 186–198. Springer (2005)
10. Paul, S.: Towards Automating the Construction & Maintenance of Attack Trees: a Feasibility Study. In: GraMSec 2014. EPTCS, vol. 148, pp. 31–46 (2014)
11. Pinchinat, S., Acher, M., Vojtisek, D.: ATSyRa: An Integrated Environment for Synthesizing Attack Trees. In: GraMSec 2015. LNCS, vol. 9390, pp. 97–101. Springer (2015)