Attack Trees in Isabelle

Florian Kammüller

Middlesex University London and
Technische Universität Berlin
f.kammueller@mdx.ac.uk

Abstract. In this paper, we present a proof theory for attack trees. Attack trees are a well established and useful model for the construction of attacks on systems since they allow a stepwise exploration of high level attacks in application scenarios. Using the expressiveness of Higher Order Logic in Isabelle, we succeed in developing a generic theory of attack trees with a state-based semantics based on Kripke structures and CTL. The resulting framework allows mechanically supported logic analysis of the meta-theory of the proof calculus of attack trees and at the same time the developed proof theory enables application to case studies. A central correctness and completeness result proved in Isabelle establishes a connection between the notion of attack tree validity and CTL. The application is illustrated on the example of a healthcare IoT system and GDPR compliance verification.

1 Introduction

Attack trees are an intuitive and practical formal method to analyse and quantify attacks on security and privacy. They are very useful to identify the steps an attacker takes through a system when approaching the attack goal. In this paper, we provide a proof calculus to analyse concrete attacks using a notion of attack validity. We define a state based semantics with Kripke models and the temporal logic CTL in the proof assistant Isabelle [1] using its Higher Order Logic (HOL)[1]. We prove the correctness and completeness (adequacy) of attack trees in Isabelle with respect to the model. This generic Kripke model enriched with CTL does not use an action based model contrary to the main stream. Instead, our model of attack trees leaves the choice of the actor and action model to the application. Nevertheless, using the genericity of Isabelle, proofs and concepts of attack trees carry over to the application model.

There are many approaches to provide a mathematical and formal semantics as well as constructing verification tools for attack trees but we pioneer the use of a Higher Order Logic (HOL) tool like Isabelle that allows proof of meta-theory – like adequacy of the semantics – and verification of applications – while being ensured that the formalism is correct.

Attack trees have been investigated on a theoretical level quite intensively; various extensions exist, e.g., to attack-defense trees and probabilistic or timed

[1] In the following, we refer to Isabelle/HOL simply as Isabelle.

attack trees. This paper uses preliminary work towards an Isabelle proof calculus for attack trees presented at a workshop [2] but accomplishes the theoretical foundation by defining a formal semantics and providing the proof of correctness and completeness and thereby establishing a feasible link for application verification. The novelty of this proof theoretic approach to attack tree verification is to take a logical approach from the very beginning by imposing the rigorous expressive Isabelle framework as the technical and semantical spine. This approach brings about a decisive advantage which is beneficial for a successful application of the attack tree formalism and consequently also characterizes our contribution: meta-theory and application verification are possible simultaneously. Since Higher Order Logic allows expressing concepts like attack trees within the logic, it enables reasoning *about* objects like attack trees, Kripke structures, or the temporal logic CTL in the logic (meta-theory) while at the same time *applying* these formalised concepts to applications like infrastructures with actors and policies (object-logics).

This paper presents the following contributions.

- We provide a proof calculus for attack trees that entails a notion of refinement of attack trees and a notion of valid attack trees.
- Validity of attack trees can be characterized by a recursive function in Isabelle which facilitates evaluation and permits code generation.
- The main theorems show the correctness and completeness of attack tree validity with respect to the state transition semantics based on Kripke structures and CTL. This meta-theorem not only provides a proof for the concepts but is part of the proof calculus for applications.
- The Isabelle attack tree formalisation is applied to the case study of formalising GDPR properties over infrastructures.

In this paper, we first introduce the underlying Kripke structures and CTL logic (Section 2). Next, we present attack trees and their notion of refinement (Section 3). The notion of validity is given by the proof calculus in Section 4 followed by the central theorem of correctness and completeness (adequacy) of attacks in Section 5 including a high level description of the proof. Section 6 shows how the framework is applied to analyse an IoT healthcare system and Section 7 extends by labelled data to enable GDPR compliance verification. We then discuss, consider related work, and draw conclusions (Section 8). All Isabelle sources are available online [3].

2 Kripke Structures and CTL in Isabelle

Isabelle is a generic Higher Order Logic (HOL) proof assistant. Its generic aspect allows the embedding of so-called object-logics as new theories on top of HOL. An Isabelle theory introduces new types, constants, and definitions. Similar to a programming language, keywords indicate these items: for example, the keyword `datatype` marks the beginning of a new type definition or `definition` introduces a new constant and its definition. In this paper, we will provide more detailed explanation of the concrete syntax when and where it is used. Object-logics, when added to Isabelle using constant and type definitions, constitute a so-called *conservative extension*. This means that no inconsistency can be introduced; conceptually, new types are defined as subsets of existing types and properties are proved using a one-to-one relationship to the new type from properties of the existing type. New properties within a theory can be subsequently proved in interaction with the user over any model defined in terms of the new types, constants, and definitions. These properties are introduced with the keyword `lemma` or `theorem` depending on their significance. Following the statement of such a property, the Isabelle tool expects the user to provide step-by-step instructions how to prove the property using existing lemmas and theorems from underlying theories or previously proved properties. In principle, proof can be a tedious process and requires expert knowledge. However, there are sophisticated proof tactics available to support reasoning: simplification, first-order resolution, and special macros to support arithmetic amongst others. The use of HOL has the advantage that it enables expressing even the most complex application scenarios, conditions, and logical requirements and HOL simultaneously enables the analysis of the meta-theory.

In this work, we make additional use of the class concept of Isabelle that allows an abstract specification of a set of types and properties to be instantiated later. We use it to abstract from states and state transition in order to create a generic framework for Kripke structures, CTL, and attack trees. Using classes the framework can then be applied to arbitrary object-logics that have a notion of state and state transition by instantiation. Isabelle attack trees have been designed as a generic framework meaning that the formalised theories can be applied to various applications. Figure 1 illustrates how the Isabelle theories in our framework are embedded into each other.

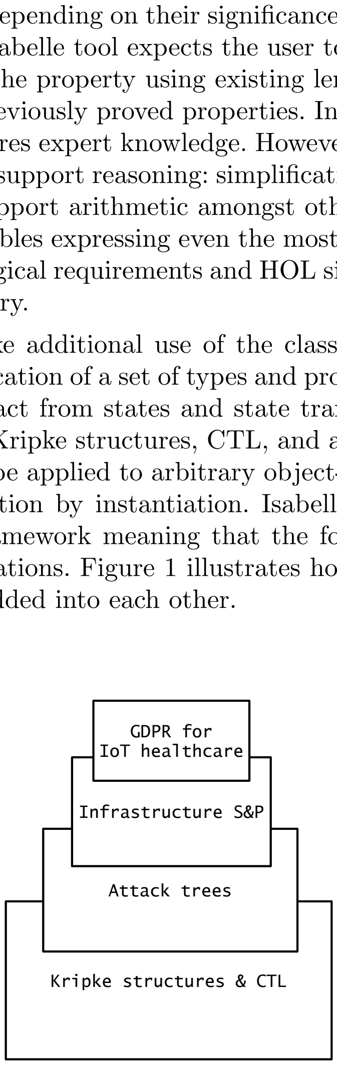

Fig. 1. Generic framework for attack trees embeds applications.

2.1 Kripke Structures and CTL

The expressiveness of Higher Order Logic (HOL) allows formalizing the notion of Kripke structures as sets of states and a generic transition relation over those in Isabelle. In addition, the branching time temporal logic CTL is embedded conservatively into HOL using Isabelle's fixpoint definitions for the CTL operators [?]. We apply Kripke structures and CTL to model state based systems and analyse properties under dynamic state changes. Snapshots of systems are the states on which we define a state transition. Temporal logic is then employed to express and prove security and privacy properties over these system models.

In Isabelle, the system states and their transition relation are defined as a class called `state` containing an abstract constant `state_transition`. It introduces the syntactic infix notation `I → I'` to denote that system state `I` and `I'` are in this relation over an arbitrary (polymorphic) type σ.

```
class state =
fixes state_transition :: [σ :: type, σ] ⇒ bool ("_  → _")
```

The above class definition introduces a new type class in Isabelle. This class `state` lies in the base class `type` which is encoded by the type judgment `:: type` following σ. All types in class `state` are characterized by having a constant called `state_transition`, with concrete infix syntax $\to$. The σ is a polymorphic type variable used here to represent an arbitrary type σ in the class `state` imposing that the state transition must be a predicate over two elements of σ, that is, a relation.

This type class `state` lifts Kripke structures and CTL to a general level allowing various instantiations to concrete state transition relations that will be provided using inductive definitions. The definition of such an inductive relation is given by a set of specific rules which are, however, part of an application like infrastructures (Section 6). Branching time temporal logic CTL is defined in general over Kripke structures with arbitrary state transitions and can later be applied to suitable theories, like infrastructures.

Based on the generic state transition $\to$ of the type class `state`, the CTL-operators EX and AX express that property f holds in some or all next states, respectively.

```
AX f ≡ { s. {f0. s → f0 } ⊆ f }
EX f ≡ { s. ∃ f0 ∈ f. s → f0 }
```

The CTL formula AG f means that on all paths branching from a state s the formula f is always true (G stands for 'globally'). It can be defined using the Tarski fixpoint theory by applying the greatest fixpoint operator.

```
AG f ≡ gfp(λ Z. f ∩  AX Z)
```

The function input to `gfp` is from a set of states `Z` to the set of states f ∩ AX `Z` transforms properties to properties – a so-called predicate transformer.

In a similar way, the other CTL operators are defined. The formal Isabelle definition of what it means that formula f holds in a Kripke structure `M` can be stated as: the initial states of the Kripke structure `init M` need to be contained in the set of all states `states M` that imply f. This is stated in the definition of the operator `check` with infix syntax ⊢.

```
M ⊢ f ≡  init M ⊆ { s ∈ states M. s ∈ f  }
```

The left side of a definition fixes parameters, here, a Kripke structure `M` and a set of states f, which can be used on the right side of the ≡ to define its meaning. In this definition, we use the set theory operators for subset relation ⊆, set membership ∈, and set collection `{ x. P x }` denoting the set of all `x` with property `P` for any predicate `P`. These set notations are provided in the rich Isabelle theory database. In an application, the set of states of the Kripke structure will be defined as the set of states reachable by the infrastructure state transition from some initial state, say `example_scenario`.

```
example_states ≡ { I. example_scenario →^*  I }
```

The relation `→^*` is the reflexive transitive closure `(_)^*` – a generic operator supplied by the Isabelle theory library – applied to the relation →. Again using here the generic theory library, automatically provides a realm of theorems about the reflexive transitive closure and powerful automated proof support for our application.

The `Kripke` constructor combines a state set, here the one of our dummy example, a set of initial states, here just the singleton set containing `example_scenario`, and a state transition relation, here →, into a Kripke structure that we name here `example_Kripke`.

```
example_Kripke ≡ Kripke example_states {example_scenario} →
```

In Isabelle, the concept of sets and predicates coincide[2]. Thus a `property` is a predicate over states which is equal to a set of states. For example, we can then try to prove that there is a path (E) to a state in which the property eventually holds (in the Future) by starting the following proof in Isabelle.

```
example_Kripke ⊢  EF property
```

Since `property` is a set of states, and the temporal operators are predicate transformers, that is, transform sets of states to sets of states, the resulting EF `property` is also a set of states – and hence again a property.

3 Attack Trees and Refinement

Attack Trees [4] are a graphical language for the analysis and quantification of attacks. If the root represents an attack, its children represent the sub-attacks. Leaf nodes are the basic attacks; other nodes of attack trees represent sub-attacks. Sub-attacks can be alternatives for reaching the goal (disjunctive node) or they must all be completed to reach the goal (conjunctive node). Figure 2 is an example of an attack tree taken from a textbook [4] illustrating the attack

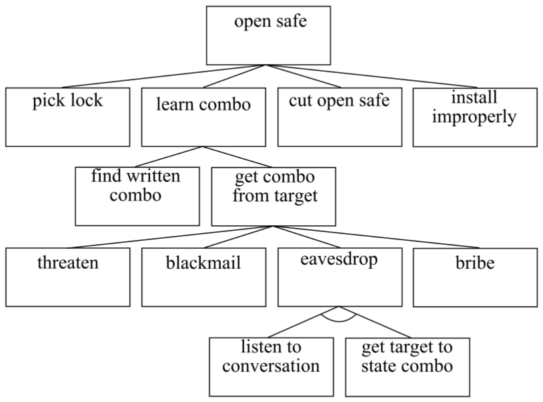

Fig. 2. Attack tree example illustrating disjunctive nodes for alternative attacks refining the attack "open safe". Near the leaves there is also a conjunctive node "eavesdrop".

of opening a safe. Nodes can be adorned with attributes, for example costs of attacks or probabilities which allows quantification of attacks (not used in the example).

3.1 Attack Tree Datatype in Isabelle

The following datatype definition `attree` defines attack trees. Isabelle allows recursive datatype definitions similar to the programming languages Haskell or ML. A datatype is given by a "`|`" separated sequence of possible cases each of which consists of a constructor name, the types of inputs to this constructor, and optionally a pretty printing syntax definition. The simplest case of an attack tree is a base attack. The principal idea is that base attacks are defined by a pair of state sets representing the initial states and the *attack property* – a set of states characterized by the fact that this property holds for them. Attacks can also be combined as the conjunction or disjunction of other attacks. The operator $\oplus_\vee$ creates or-trees and $\oplus_\wedge$ creates and-trees. And-attack trees $l \oplus_\wedge^s$ and or-attack trees $l \oplus_\vee^s$ consist of a list of sub-attacks – again attack trees.

```
 datatype (σ :: state)attree =
  BaseAttack (σ set)×(σ set) ("𝒩 (_)")
| AndAttack (σ attree)list (σ set)×(σ set) ("_ ⊕∧^(-)")
| OrAttack  (σ attree)list (σ set)×(σ set) ("_ ⊕∨^(-)")
```

The attack goal is given by the pair of state sets on the right of the operator $\mathcal{N}$, $\oplus_\vee$ or $\oplus_\wedge$, respectively. A corresponding projection operator is defined as the function `attack`.

[2] In general, this is often referred to as *predicate transformer semantics.*

```
primrec attack :: (σ::state)attree ⇒ (σ set)×(σ set)
where
  attack (BaseAttack b) = b
| attack (AndAttack as s) = s
| attack (OrAttack as s) = s
```

Functions over datatypes can be given with `primrec` which enables defining an operator, here `attack`, by listing the possible cases and describing the semantics using simple equations and pattern matching on the left side.

3.2 Attack Tree Refinement

When we develop an attack tree, we proceed from an abstract attack, given by an attack goal, by breaking it down into a series of sub-attacks. This proceeding corresponds to a process of *refinement*. Therefore, as part of the attack tree calculus, we provide a notion of attack tree refinement. This can be done elegantly by defining an infix operator $\sqsubseteq$. The intuition of developing an attack tree from the root to the leaves is illustrated in Figure 3. The example attack tree on the left side has a leaf that is expanded by the refinement into an and-attack with two steps. Formally, we define the semantics of the refinement operator by an

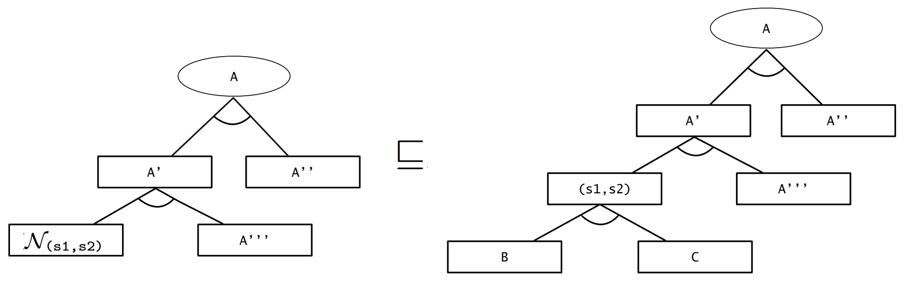

Fig. 3. Attack tree example illustrating refinement of an and-subtree.

inductive definition for the constant $\sqsubseteq$, that is, the smallest predicate closed under the set of specified rules (see Table **??**).

```
inductive refines_to :: [(σ :: state) attree, σ attree] ⇒ bool ("_ ⊑ _")
where
  refI: ⟦  A = (l @ [𝒩(s1,s2)] @ l'')⊕∧^(s0,s3); A' = l' ⊕∧^(s1,s2);
          A'' = l @ l' @ l'' ⊕∧^(s0,s3) ⟧ ⟹ A ⊑ A''
| ref_or: ⟦ as ≠ []; ∀ A' ∈ set(as). A ⊑ A' ∧ attack A = s
          ⟧ ⟹ A ⊑ as ⊕∨s
| ref_trans: ⟦ A ⊑ A'; A' ⊑ A'' ⟧ ⟹ A ⊑ A''
| ref_refl : A ⊑ A
```

The rule `refI` captures the intuition expressed in Figure 3: a sequence of leaves in an and-subtree can be refined by replacing a single leaf by a new subsequence (the `@` is the list append in Isabelle). Rule `ref_or` describes or-attack refinement. To refine a node into an or-attack, all sub-trees in the or-attack list need to refine the parent node. The remaining rules define $\sqsubseteq$ as a pre-order on sub-trees of an attack tree: it is reflexive and transitive.

Refinement of attack trees defines the stepwise process of expanding abstract attacks into more elaborate attacks only syntactically. There is no guarantee that the refined attack is possible if the abstract one is, nor vice-versa. We need to provide a semantics for attacks in order to judge whether such syntactic refinements represent possible attacks. To this end, we now formalise the semantics of attack trees by a proof theory.

4 Proof Calculus

A valid attack, intuitively, is one which is fully refined into fine-grained attacks that are feasible in a model. The general model we provide is a Kripke structure, i.e., a set of states and a generic state transition. Thus, feasible steps in the model are single steps of the state transition. We call them valid base attacks. The composition of sequences of valid base attacks into and-attacks yields again valid attacks if the base attacks line up with respect to the states in the state transition. If there are different valid attacks for the same attack goal starting from the same initial state set, these can be summarized in an or-attack.

```
 fun is_attack_tree :: [(σ :: state) attree] ⇒ bool  ("⊢_")
 where
  att_base:  ⊢ $\mathcal{N}_s$ = ∀ x ∈ fst s. ∃ y ∈ snd s. x  → y
| att_and: ⊢ (As :: (σ::state attree list)) $\oplus_\wedge^s$ =
          case As of
            [] ⇒ (fst s ⊆ snd s)
          |  [a] ⇒ ⊢ a ∧ attack a = s
          |  a # l ⇒ ⊢ a ∧ fst(attack a) = fst s
                       ∧ ⊢ l $\oplus_\wedge^{(snd(attack\ a),snd(s))}$
| att_or: ⊢ (As :: (σ::state attree list)) $\oplus_\vee^s$ =
         case As of
            [] ⇒ (fst s ⊆ snd s)
         | [a] ⇒ ⊢ a ∧ fst(attack a) ⊇ fst s ∧ snd(attack a) ⊆ snd s
         | a # l ⇒ ⊢ a ∧ fst(attack a) ⊆ fst s ∧ snd(attack a) ⊆ snd s
                     ∧ ⊢ l $\oplus_\vee^{(fst\ s\ -\ fst(attack\ a),snd\ s)}$
```

More precisely, the different cases of the validity predicate are distinguished by pattern matching over the attack tree structure.

- A base attack $\mathcal{N}_{(s0,s1)}$ is valid if from all states in the pre-state set `s0` we can get with a single step of the state transition relation to a state in the post-state set `s1`. Note, that it is sufficient for a post-state to exist for each pre-state. After all, we are aiming to validate attacks, that is, possible attack paths to some state that fulfills the attack property.

- An and-attack `As` $\oplus_{\wedge}^{(\texttt{s0},\texttt{s1})}$ is a valid attack if either of the following cases holds:
  - empty attack sequence `As`: in this case all pre-states in `s0` must already be attack states in `s1`, i.e., `s0` $\subseteq$ `s1`;
  - attack sequence `As` is singleton: in this case, the singleton element attack `a` in `[a]`, must be a valid attack and it must be an attack with pre-state `s0` and post-state `s1`;
  - otherwise, `As` must be a list matching `a # l` for some attack `a` and tail of attack list `l` such that `a` is a valid attack with pre-state identical to the overall pre-state `s0` and the goal of the tail `l` is `s1` the goal of the overall attack. The pre-state of the attack represented by `l` is `snd(attack a)` since this is the post-state set of the first step `a`.
- An or-attack `As` $\oplus_{\vee}^{(s0,s1)}$ is a valid attack if either of the following cases holds:
  - the empty attack case is identical to the and-attack above: `s0` $\subseteq$ `s1`;
  - attack sequence `As` is singleton: in this case, the singleton element attack `a` must be a valid attack and its pre-state must include the overall attack pre-state set `s0` (since `a` is singleton in the or) while the post-state of `a` needs to be included in the global attack goal `s1`;
  - otherwise, `As` must be a list `a # l` for an attack `a` and a list `l` of alternative attacks. The pre-states can be just a subset of `s0` (since there are other attacks in `l` that can cover the rest) and the goal states `snd(attack a)` need to lie all in the overall goal state set `s1`. The other or-attacks in `l` need to cover only the pre-states `fst s - fst(attack a)` (where `-` is set difference) and have the same goal `snd s`.

The proof calculus is thus completely described by one recursive function. This is a major improvement to the inductive definition provided in the preliminary workshop paper [2] that inspired this paper. Our notion of attack tree validity is more concise hence less prone to stating inconsistent definitions and still allows to infer properties important for proofs. The increase of consistency is because other important or useful algebraic properties can be derived from the recursive function definition. Note, that preliminary experiments on a proof calculus for attack trees in Isabelle [2] used an inductive definition that had a larger number of rules than the three cases we have in our recursive function definition `is_attack_tree`. The earlier inductive definition integrated a fair number of properties as inductive rules which are now proved from the three cases of `is_attack_tree`.

It might appear that Kripke semantics interprets conjunction as sequential (ordered) conjunction instead of parallel (unordered) conjunction. However, this is not the case: the ordering of events or actions is implicit in the states. Therefore, any kind of interleaving (or true parallelism) of state changing actions is possible. This is inserted as part of the application – for example in the Infrastructures definition of the state transition in Section 6. There the order of actions between states depends on the pre-states and post-states only.

Given the proof calculus, the notion of validity of an attack tree can be used to identify valid refinements already at a more abstract level. The notion $\sqsubseteq_V$

denotes that the refinement of the attack tree on the left side is to a valid attack tree on the right side.

```
A ⊑_V A' ≡ ( A ⊑ A' ∧ ⊢ A')
```

Taking this one step further, we can say that an abstract attack tree is valid if there is a valid attack tree it refines to.

```
⊢_V A ≡ (∃ A'. A ⊑_V A')
```

Thereby, we have achieved what we initially wanted: to state that an abstract attack tree `A` is actually a valid attack tree, we can conjecture $\vdash_V$`A`. This results in the proof obligation of finding a valid attack tree ⊢`A'` such that `A` ⊑`A'`. For practical purposes, the following lemma implements this method.

lemma `ref_valI: A ⊑ A' ⟹ ⊢ A' ⟹ ⊢_V A`

We are going to use this method on the case study in Section 6.2 for the attack tree analysis.

5 Correctness and Completeness of Attack Trees

The novel contribution of this paper is to equip attack trees with a Kripke semantics. Thereby, a valid attack tree corresponds to an attack sequence. The following correctness theorem provides this: if `A` is a valid attack on property `s` starting from initial states described by `I`, then from all states in `I` there is a path to the set of states fulfilling `s` in the corresponding Kripke structure.

```
theorem AT_EF: ⊢ A :: (σ :: state) attree) ⟹ (I, s) = attack A
⟹ Kripke {t . ∃ i ∈ I. i →^* t} I ⊢ EF s
```

It is not only an academic exercise to prove this theorem. Since we use an embedding of attack trees into Isabelle, this kind of proof about the embedded notions of attack tree validity ⊢ and CTL formulas like `EF` is possible. At the same time, the established relationship between these notions can be applied to case studies. Consequently, if we apply attack tree refinement to spell out an abstract attack tree for attack `s` into a valid attack sequence, we can apply theorem `AT_EF` and can immediately infer that `EF s` holds.

Theorem `AT_EF` also extends to validity of abstract attack trees. That is, if an "abstract" attack tree `A` can be refined to a valid attack tree, correctness in CTL given by `AT_EF` applies also to the abstract tree.

```
theorem ATV_EF: ⊢_V A :: (σ :: state) attree) ⟹ (I, s) = attack A
⟹ Kripke {t . ∃ i ∈ I. i →^* t} I ⊢ EF s
```

The inverse direction of theorem `AT_EF` is a completeness theorem: if states described by predicate `s` can be reached from a finite nonempty set of initial states `I` in a Kripke structure, then there exists a valid attack tree for the attack `(I,s)`.

```
theorem Completeness: I ≠ {} ⟹ finite I ⟹
Kripke {t . ∃ i ∈ I. i →^* t} I ⊢  EF s
⟹ ∃ A :: (σ::state)attree. ⊢ A ∧ (I, s) = attack A
```

Correctness and Completeness are proved in Isabelle within the theory `AT.thy` [3]. The interactive proofs including auxiliary lemmas consist of nearly 1200 lines of proof commands. However, we have proved these theorems once for all. Owing to the modular organisation of our framework they are meta-theoretic theorems usable for any object logic that models an application.

6 Application to Infrastructures, Policies, and Actors

The Isabelle Infrastructure framework supports the representation of infrastructures as graphs with actors and policies attached to nodes. These infrastructures are the *states* of the Kripke structure.

The transition between states is triggered by non-parametrized actions `get`, `move`, `eval`, and `put` executed by actors. Actors are given by an abstract type `actor` and a function `Actor` that creates elements of that type from identities (of type `string`). Policies are given by pairs of predicates (conditions) and sets of (enabled) actions.

```
type_synonym policy = ((actor ⇒ bool) × action set)
```

Actors are contained in an infrastructure graph.

```
datatype igraph = Lgraph (location × location)set
                         location ⇒ identity set
                         actor ⇒ (string set × string set)
                         location ⇒ (string × acond)
```

An `igraph` has just one constructor function `Lgraph`. It constructs an `igraph` from a set of location pairs representing the topology of the infrastructure as a graph of nodes and a list of actor identities associated to each node (location) in the graph. The third component of an `igraph` associates actors to a pair of string sets by a pair-valued function whose first range component is a set describing the credentials in the possession of an actor and the second component is a set defining the roles the actor can take on. More importantly in this context, the fourth component of an `igraph` assigns locations to a pair of a string that defines the state of the component and an element of type `acond`. This type `acond` is defined as a set of labelled data representing a condition on that data. Corresponding projection functions for each of these components of an `igraph` are provided; they are named `gra` for the actual set of pairs of locations, `agra` for the actor map, `cgra` for the credentials, and `lgra` for the state of a location and the data at that location.

Infrastructures are given by the following datatype that contains an infrastructure graph of type `igraph` and a policy given by a function that assigns local policies over a graph to all locations of the graph.

```
datatype infrastructure = Infrastructure igraph
                                        igraph ⇒ location ⇒ policy set
```

There are projection functions `graphI` and `delta` when applied to an infrastructure return the graph and the policy, respectively. Policies specify the expected behaviour of actors of an infrastructure. They are defined by the `enables` predicate: an actor `h` is enabled to perform an action `a` in infrastructure `I`, at location `l` if there exists a pair `(p,e)` in the local policy of `l` (`delta I l` projects to the local policy) such that the action `a` is a member of the action set `e` and the policy predicate `p` holds for actor `h`.

```
enables I l h a ≡ ∃ (p,e) ∈ delta I l. a ∈ e ∧ p h
```

We now flesh out the abstract state transition introduced in Section 2.1 by defining an inductive relation $\rightarrow_n$ for state transition between infrastructures. This state transition relation is dependent on actions but also on enabledness and the current state of the infrastructure. For illustration purposes we consider the rule for `get_data` only (see the complete source code [3] for full details of other rules)[3].

```
get_data : G = graphI I ⟹ a @_G l ⟹
           enables I l' (Actor a) get ⟹
           I' = Infrastructure
                 (Lgraph (gra G)(agra G)(cgra G)
                         ((lgra G)(l := (fst (lgra G l),
                                   snd (lgra G l) ∪ {new}))))
                         (delta I)
           ⟹ I →_n I'
```

The new state `I'` of the infrastructure can be reached from `I` if the actor `h` is in location `l`, action `get` is enabled for `h` at location `l`. Under those preconditions, data `new` can be added to the actor's current location `l` formalised here using the function update `:=` for the fourth component `lgra G` of the infrastructure's `igraph`.

6.1 Application Example from IoT Healthcare

The example of an IoT healthcare systems is from the CHIST-ERA project SUCCESS [5] on monitoring Alzheimer's patients. Figure 4 illustrates the system architecture where data collected by sensors in the home or via a smart phone helps monitoring bio markers of the patient. The data collection is in a cloud based server to enable hospitals (or scientific institutions) to access the data which is controlled via the smart phone. We show the encoding of the `igraph` for this system architecture in the Infrastructure model.

```
ex_graph ≡ Lgraph {(home, cloud), (sphone, cloud), (cloud,hospital)}
                  (λ x. if x = home then {''Patient''}
                        else (if x = hospital then {''Doctor''} else {}))
                        ex_creds ex_locs
```

[3] We deliberately omit here the DLM constraints for illustration purposes (see below)

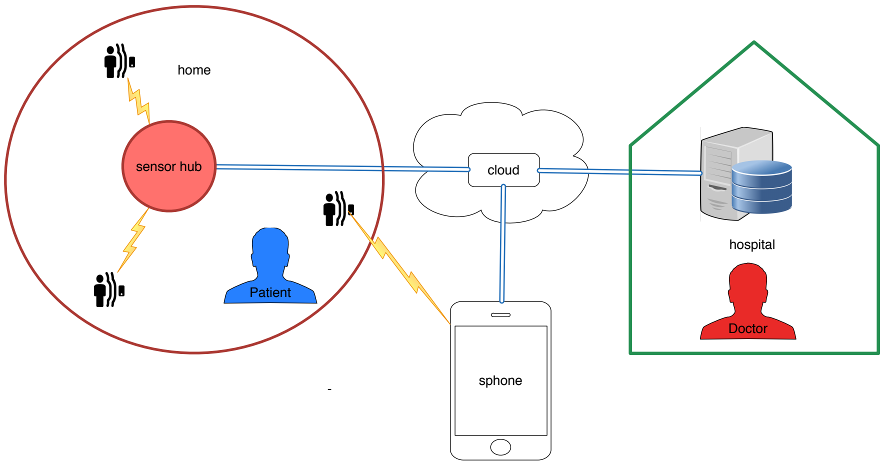

Fig. 4. IoT healthcare monitoring system for SUCCESS project

The identities `Patient` and `Doctor` represent patients and their doctors; double quotes `''s''` indicate strings in Isabelle/HOL. The global policy is 'only the patient and the doctor can access the data in the cloud':

```
fixes global_policy::[infrastructure, identity] ⇒ bool
defines  global_policy I a ≡  a ∉ gdpr_actors ⟶
                ¬(enables I cloud (Actor a) get)
```

Local policies are represented as a function over an `igraph G` that additionally assigns each location of a scenario to its local policy given as a pair of requirements to an actor (first element of the pair) in order to grant him actions in the location (second element of the pair). The predicate $@_G$ checks whether an actor is at a given location in the graph G.

```
local_policies G ≡
(λ x. if x = home then {(λ y. True, {put,get,move,eval})}
 else (if x = sphone then
  {((λ y. has G (y, ''PIN'')), {put,get,move,eval})}
    else (if x = cloud then {(λ y. True, {put,get,move,eval})}
          else (if x = hospital then
                {((λ y. (∃ n. (n @_G hospital) ∧ Actor n = y ∧
                 has G (y, ''skey''))), {put,get,move,eval})} else {}))))
```

6.2 Using Attack Tree Calculus

Since we consider a predicate transformer semantics, we use sets of states to represent properties. For example, the attack property is given by the following set `sgdpr`.

```
  sgdpr ≡ {x. ¬ (global_policy x ''Eve'')}
```

The attack we are interested in is to see whether for the scenario

```
gdpr_scenario ≡  Infrastructure ex_graph local_policies
```

from the initial state `Igdpr` ≡{`gdpr_scenario`}, the critical state `sgdpr` can be reached, i.e., is there a valid attack `(Igdpr,sgdpr)`?

To set up this question as a proof goal, we can now use the meta-theory for valid abstract attack trees developed in Section 4 which allows setting out from this abstract attack.

$$\vdash_V \; [\mathcal{N}_{\texttt{(Igdpr,sgdpr)}}] \oplus_{\wedge}^{\texttt{(Igdpr,sgdpr)}}$$

We can then prove that there is a refinement to an and-attack where the set `GDPR` is an intermediate state where `Eve` accesses the cloud.

$$[\mathcal{N}_{\texttt{(Igdpr,sgdpr)}}] \oplus_{\wedge}^{\texttt{(Igdpr,sgdpr)}}$$
$$\sqsubseteq$$
$$[\mathcal{N}_{\texttt{(Igdpr,GDPR)}}, \mathcal{N}_{\texttt{(GDPR,sgdpr)}}] \oplus_{\wedge}^{\texttt{(Igdpr,sgdpr)}}$$

We can then finish the proof by deriving that the second refined attack is a valid and-attack using the attack tree proof calculus.

$$\vdash \; [\mathcal{N}_{\texttt{(Igdpr,GDPR)}}, \mathcal{N}_{\texttt{(GDPR,sgdpr)}}] \oplus_{\wedge}^{\texttt{(Igdpr,sgdpr)}}$$

For the Kripke structure

```
gdpr_Kripke ≡ Kripke { I. gdpr_scenario →* I } Igdpr
```

we can alternatively apply the Correctness theorem `AT_EF` to immediately derive from the previous result the following CTL statement.

```
gdpr_Kripke ⊢  EF sgdpr
```

This application of the meta-theorem of Correctness of attack trees saves us proving the CTL formula tediously by exploring the state space. However, a more convincing case for using the Correctness and Completeness meta-theorems is given next when we consider how to improve the access control on data to guarantee GDPR level security and privacy.

7 Data Protection by Design for GDPR compliance

7.1 General Data Protection Regulation (GDPR)

On 26th May 2018, the GDPR has become mandatory within the European Union and hence also for any supplier of IT products. Breaches of the regulation will be fined with penalties of 20 Million EUR. For this paper, we use the final proposal [6] as our source. Despite the relatively large size of the document of 209 pages, the technically relevant portion for us is only about 30 pages (Pages 81–111, Chapters I to Chapter III, Section 3). In summary, Chapter III specifies that the controller must give the data subject *read access* (1) to any information, communications, and "meta-data" of the data, e.g., retention time and purpose. In addition, the system must enable *deletion of data* (2) and restriction of processing.

An invariant condition for data processing resulting from these Articles is that the system *functions* must *preserve* any of the access rights of personal data (3).

7.2 Security and Privacy by Labeling Data

The Decentralised Label Model (DLM) [7] introduced the idea to label data by owners and readers. We pick up this idea and formalize a new type to encode the owner and the set of readers.

```
type_synonym dlm = actor × actor set
```

Labelled data is then just given by the type `dlm × data` where `data` can be any data type. Additional meta-data, like retention time and purpose, can be encoded as part of this type `data`. We omit these detail here for conciseness of the exposition.

Using labeled data, we can now express the essence of Article 4 Paragraph (1): 'personal data' means any information relating to an identified or identifiable natural person ('data subject'). Since we have a more constructive system view, we express this by defining the owner of a data item `d` of type `dlm` as the actor that is the first element in the pair that is the first of the pair `d`. Then, we use this function to express the predicate "owns".

```
definition owner :: dlm × data ⇒ actor
where owner d ≡ fst(fst d)

definition owns :: [igraph, location, actor, dlm × data] ⇒ bool
where owns G l a d ≡ owner d = a
```

The introduction of a similar function for readers projecting the second element of a `dlm` label

```
definition readers :: dlm × data ⇒ actor set
where readers d ≡ snd (fst d)
```

enables specifying whether an actor may access a data item.

```
definition has_access :: [igraph, location, actor, dlm × data] ⇒ bool
where has_access G l a d ≡ owns G l a d ∨ a ∈ readers d
```

For our example of an IoT health care monitoring system, the data and its privacy access control definition is given by the parameter `ex_locs` specifying that the data 42, for example some bio marker's value, is located in the cloud, is owned by the patient, and can be read by the doctor (`''free''` is the state of the cloud component).

```
ex_locs ≡ (λ x. if x = cloud
           then (''free'', {((Actor ''Patient'',{Actor ''Doctor''}),42)})
           else ('''',{}))
```

7.3 Privacy Preserving Functions

The labels of data must not be changed by processing: we have identified this finally as an invariant (3) resulting from the GDPR in Section 7. This invariant can be formalized in our Isabelle model by a type definition of functions on labeled data that preserve their labels.

```
typedef label_fun = {f :: dlm×data ⇒ dlm×data. ∀ x. fst x = fst (f x)}
```

We also define an additional function application operator ⇕ on this new type. Then we can use this restricted function type to implicitly specify that only functions preserving labels may be applied in the definition of the system behaviour in the state transition rule for action `eval` (see [3]).

7.4 Policy Enforcement

We can now use the labeled data to encode the privacy constraints of the GDPR in the rules. For example, the `get_data` rule has now labelled data `((Actor a', as), n)` and used the labeling in the precondition to guarantee that only entitled users can get data: `Actor a` has to be in the set of readers `as` to have this data item added to his location `l`.

```
get_data : G = graphI I ⟹ a @_G l ⟹ enables I l' (Actor a) get ⟹
           ((Actor a', as), n) ∈ snd (lgra G l') ⟹ Actor a ∈ as ⟹
           I' = Infrastructure
                 (Lgraph (gra G)(agra G)(cgra G)
                         ((lgra G)(l := (fst (lgra G l),
                             snd (lgra G l) ∪ {((Actor a', as), new)}))))
                         (delta I)
           ⟹ I →_n I'
```

Using the formal model of infrastructures, we can now prove privacy by design for GDPR compliance of the specified system. We can show how the properties relating to data ownership, processing and deletion can be formally captured using Kripke structures and CTL and the Infrastructure framework. As an example, consider the preservation of data ownership.

Processing preserves privacy We can prove that processing preserves ownership as defined in the initial state for all paths globally (AG) within the Kripke structure and in all locations of the graph.

```
theorem GDPR_three: h ∈ gdpr_actors ⟹ l ∈ gdpr_locations ⟹
  owns (Igraph gdpr_scenario) l (Actor h) d ⟹
  gdpr_Kripke ⊢
    AG {x. ∀ l ∈ gdpr_locations. owns (Igraph x) l (Actor h) d }
```

Note, that it would not be possible to express this property in Modelcheckers (let alone prove it) since they only allow propositional logic within states. This generalisation is only possible since we use Higher Order Logic. The proof of this property is straightforward evaluation of the CTL rules.

Applying Correctness to prove Absence of Attacks The contraposition of the Correctness theorem grants that if (EF f) does *not* hold in a Kripke structure, then there is *no* attack `(I,`f`)` for the initial states of the Kripke structure `I`. Since properties are expressed as sets, negation is expressed in the theorem by using the set complement `-P` for the negation of property `P`.

```
h ∈ gdpr_actors ⟹ l ∈ gdpr_locations ⟹
owns (Igraph gdpr_scenario) l (Actor h) d ⟹
attack A = (Igdpr, -{x. ∀ l ∈ gdpr_locations.
                    owns (Igraph x) l (Actor h) d })
⟹ ¬(⊢ A::infrastructure attree)
```

Proving the absence of attacks for attack trees is in general very difficult but becomes feasible owing to the meta-theorem Correctness and the possibility to interleave meta-theoretic reasoning with that in the object logic in Isabelle.

8 Conclusions

In this paper, we have presented a proof theory for attack trees in Isabelle's Higher Order Logic (HOL). We have shown the incremental and generic structure of this framework, presented correctness and completeness results equating valid attacks to EF s formulas. The proof theory has been illustrated on an IoT healthcare infrastructure where the meta-theorem of completeness could be directly applied to infer the existence of an attack tree from CTL. The practical relevance has been demonstrated on GDPR compliance verification.

There are excellent foundations available based on graph theory [8]. They provide a very good understanding of the formalism, various extensions (like attack-defense trees [9]) and differentiations of the operators (like sequential conjunction (SAND) versus parallel conjunction [10]) and are amply documented in the literature. These theories for attack trees provide a thorough foundation for the formalism and its semantics. The main problem that adds complexity to the semantical models is the abstractness of the descriptions in the nodes. This leads to a variety of approaches to the semantics, e.g. propositional semantics, multiset semantics, and equational semantics for ADtrees [9]. The theoretical foundations allow comparison of different semantics, and provide a theoretical framework to develop evaluation algorithms for the quantification of attacks.

More practically oriented formalisations, e.g. [11], focus on an action based-approach where the attack goals are represented as labels of attack tree nodes which are actions that an attacker has to execute to arrive at the goal.

A notable exception that uses, like our approach, a state based semantics for attack trees is the recent work [12]. However, this work is aiming at assisted generation of attack trees from system models. The tool ATSyRA supports this process. The paper [12] focuses on describing a precise semantics of attack tree in terms of transition systems using "under-match", "over-match", and "match" to arrive at a notion of correctness. In comparison, we use additionally CTL logic to describe the correctness relation precisely. Also we use a fully formalised and proved Isabelle model.

Surprisingly, the use of an automated proof assistant, like Isabelle, has not been considered before despite its potential of providing a theory and analysis of attacks simultaneously. The essential attack tree mechanism of disjunction and conjunction in tree refinement is relatively simple. The complexity in the theories is caused by the attempt to incorporate semantics to the attack nodes and relate the trees to actual scenarios. This is why we consider the formalisation of a foundation of attack trees in the interactive prover Isabelle since it supports logical modeling and definitions of datatypes very akin to algebraic specification but directly supported by semi-automated analysis and proof tools.

The workshop paper [2] has inspired the present work but is vastly superseded by it. The novelties are:

- Attack trees have a state based semantics formalised in the framework.
- Correctness and completeness are proved based on the formal semantics.
- The Isabelle framework is generic using type classes, that is, works for any state model. Infrastructures with actors and policies are an instantiation.
- The semantics, correctness and completeness theorems facilitate application verification.

References

1. T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, LNCS **2283**, Springer-Verlag, 2002.
2. F. Kammüller, "A proof calculus for attack trees," in *Data Privacy Management, DPM17, 12th Int. Workshop*, LNCS **10436**, Springer, 2017.
3. ——, "Isabelle infrastructure framework with iot healthcare s&p application," 2018, available at https://github.com/flokam/IsabelleAT.
4. B. Schneier, *Secrets and Lies: Digital Security in a Networked World.* Wiley, 2004.
5. CHIST-ERA, "Success: Secure accessibility for the internet of things," 2016, http://www.chistera.eu/projects/success.
6. E. Union, "The eu general data protection regulation (gdpr)," Accessed 20.3. 2018, proposal for a Regulation of the European Parliament and of the Council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation) [first reading] - Analysis of the final compromise text with a view to agreement, Brussels, 15 December 2015. [Online]. Available: http://www.eugdpr.org
7. A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," *IEEE Symposium on Security and Privacy.* IEEE, 1999.
8. B. Kordy, L. Piètre-Cambacédés, and P. Schweitzer, "Dag-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer Science Review*, vol. 13–14, pp. 1–38, 2014.
9. B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer, "Attack-defense trees," *Journal of Logic and Computation*, vol. 24, no. 1, pp. 55–87, 2014.
10. R. Jhawar, B. Kordy, S. Mauw, S. Radomirovic, and R. Trujillo-Rasua, "Attack trees with sequential conjunction," *IFIP SEC'15*, AICT **455**, Springer 2015.
11. Z. Aslanyan, F. Nielson, and D. Parker, "Quantitative verification and synthesis of attack-defence scenarios," *CSF'16*, IEEE 2016.
12. M. Audinot, S. Pinchinat, and B. Kordy, "Is my attack tree correct?" *ESORICS'2017*, LNCS **10492**, Springer, 2017.