Attack Trees in Isabelle extended with Probabilities for Quantum Cryptography

Florian Kammüller

Middlesex University London and Technische Universität Berlin

Abstract

In this paper, we present a proof calculus for Attack Trees and how its application to Quantum Cryptography is made possible by extending the framework to probabilistic reasoning on attacks. 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.

Furthermore in this paper, we illustrate the application of Attack Trees to security protocols on the example of the Quantum Key Distribution (QKD) algorithm. The application motivates the extension of the Attack Tree proof calculus by probabilities. We therefore introduce probabilities to quantify finite event sequences and show how this extension can be used to extend CTL to its probabilistic version PCTL. We show on the example of QKD how probabilistic reasoning with PCTL enables proof of quantitative security properties.

Keywords: Attack trees, Formal methods, Verification, Probability, Quantum Cryptography

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

Email address: `f.kammueller@mdx.ac.uk` (Florian Kammüller)

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.

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-defence trees and probabilistic or timed 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 with respect to a state transition model. This proof of adequacy guarantees not only consistency of attack trees for given systems but also provides a practical tool for application verification since it allows to transform results about attacks into temporal logic statements. 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 semantic 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).

To show the versatility of the Attack Tree proof calculus, we present here a radically different example from the existing applications to IoT and GDPR [3]. We venture into the world of security protocols: we present a first step towards a formalisation of the Quantum Key Distribution (QKD) algorithm in Isabelle. We focus on the formalisation of the main probabilistic argument why Bob cannot be certain about the key bit sent by Alice before he has a chance to compare the chosen polarization scheme. This means that any adversary Eve is in the same position as Bob and cannot be certain about the transmitted keybits. While Attack Trees can be used to show that an interception attack is possible, the real interest is to prove how likely is the attack to succeed. We additionally need probabilistic statements about attacks.

This paper presents the following contributions on attack trees.

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

- 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 enables 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. These meta-theorems not only provide a proof for the concepts but are part of the proof calculus for applications.

Apart from details on the code generation, these contributions were already presented at a conference [3].

Further contributions of this paper on the application to Quantum Cryptography subsume the earlier results and exceed them largely by

- constructing an application model for security protocols within the Isabelle framework illustrating it on the Quantum Key Distribution (QKD) algorithm,
- introducing probabilities,
- formalising the probabilistic extension PCTL of CTL,
- and illustrating how it can be applied to enable quantitative proofs for QKD.

In this paper, we first introduce the Isabelle Infrastructure framework and its formalisation of Kripke structures and the temporal logic CTL (Section 2). In this section, we pay particular attention to explaining the added value of the conservative extension for Attack Trees with respect to earlier preliminary and related work to that end (Section 2.2). 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. We next present the novel application to Quantum Cryptography in Section 6 first introducing the QKD protocol in Section 6.1, then showing how protocols and in particular QKD are formalised in our framework (Section 6.2) before illustrating attack tree analysis by refinement on the interception attack (Section 6.3). The case study motivates the extension to probabilities in Section 7. We first provide some basic probability theory in Section 7.1 before introducing the formalisation of Probabilistic CTL (Section 7.2) and showing the probabilistic model of QKD (Section 7.3) that then enables the probabilistic attack analysis (Section 7.4). We then discuss, consider related work, and draw conclusions (Section 8). All Isabelle sources are available online [4] including the generated Scala code for the attack tree checking function.

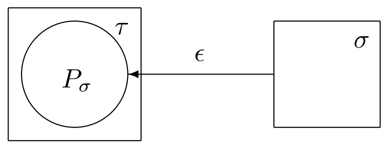

Figure 1: Conservative extension: new type σ defined as a copy of subset P_σ of existing type τ thereby making all properties over σ derivable and hence consistent with foundation τ.

2. Isabelle Infrastructure Framework, Kripke Structures, and CTL

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. 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 σ are defined as nonempty subsets P_σ of existing types and properties over σ are proved from properties of the existing type using a one-to-one relationship ϵ between σ and P_σ (see Figure 1).

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.

2.1. Isabelle Infrastructure Overview

Isabelle supports modular reasoning, that is, we can prove theorems *in* an object logic but also *about* it. This allows the building of telescope-like structures in which a meta-theory at a lower level embeds a more concrete "application" at a higher level. Properties are proved at each level. Interactive proof is used to prove these properties but the meta-theory can be applied to immediately produce results. Figure 2 gives an overview of the Isabelle Infrastructure framework with its layers of object-logics – each level below embeds the one above. The layered structure of the framework enables that the reasoning about the meta-theory at one level can be done independently and once-for all for all levels above. For example, properties relating to the definition of temporal logic operators can be performed at the lowest level but these meta-theoretical properties can then be used at all higher levels to show other results. This not only supports consistency and clarity but also the possibility to enable collaboration of experts at different levels of expertise. An Isabelle expert can continue the development and improvement of the lower levels, for example, adding properties for CTL and attack tree reasoning, thereby enhancing automated tactics, while a security domain expert can work at the higher levels on application examples, for example, testing and refining protocol definitions.

Yet, the framework stays extensible: the layer we added in this extended paper contains probability theory and the probabilistic temporal logic PCTL.

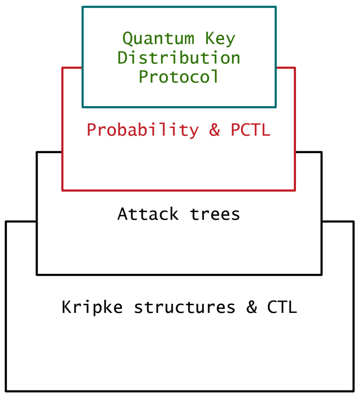

Figure 2: Generic framework for infrastructures embeds applications.

Those two theory extensions are added on top of Attack Trees but are conceptually generic framework extensions.

Moreover, Isabelle provides the possibility to generate code from executable parts of the formal models. For our infrastructure framework, we generate code in the programming language Scala for the central predicate of the attack tree proof theory (see online resources [4]). In a practical workflow, this generated code could be used to automatically check an attack tree for validity while being sure that this check is correct based on the underlying formal proofs provided in this paper. While constructing Isabelle definitions and proofs requires expert knowledge, these efforts are made once and for all to guarantee powerful meta-theoretic results – like the adequacy of the proof theory provided in this paper – and to generate code that can then be used on examples.

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.

2.2. Relation to Earlier Formalisations

The Isabelle Infrastructure framework has been created initially for the modeling and analysis of Insider threats [5]. Its use has been validated on the most well-known insider threat patterns identified by the CERT-Guide to Insider threats [6]. More recently, this Isabelle framework has been successfully applied to realistic case studies of insider attacks in airplane safety [7] and on auction protocols [8]. These larger case studies as well as complementary work on the analysis of Insider attacks on IoT infrastructures, e.g. [9], have motivated the extension of the original framework by Kripke structures and temporal logic as well as a formalisation of attack trees [3]. Recently, GDPR compliance verification has been demonstrated [10].

In the course of this extension, the Isabelle framework has been restructured such that it is now a general framework for the state-based security analysis of infrastructures with policies and actors. Temporal logic and Kripke structure build the foundation. Meta-theoretical results have been established to show equivalence between attack trees and CTL statements. This foundation provides a generic notion of state transition on which attack trees and temporal logic can be used to express properties.

An earlier workshop paper [2] already presented the idea of a a tentative proof calculus for Attack Trees in Isabelle. In comparison, the current proof calculus for attack trees provides a full formal semantics of Attack Trees in the branching time temporal logic CTL. This enables proving the Correctness and Completeness theorems presented in the current paper and thus guarantees soundness of any attacks proved in the present framework. This soundness is based on the embedding of CTL as a conservative extension of Higher Order Logic combined with the meta-theorems of Correctness and Completeness. The difference to the earlier tentative paper is apparent when considering the actual proof calculus [2, p.8]. Where in the present paper the core predicate `is_attack_tree` (see Section 4) is defined on top of the CTL validity $\vdash$, a notion of logical validity had to be introduced in the workshop paper and its semantics defined axiomatically by assuming rules ([Table 3, p.8][2]). While reasoning with these rules is possible, no properties relating to a system and its behaviour is possible. This only becomes possible by a relation to a notion of system behaviour and a corresponding logic, as has been now provided by the formal semantics of Attack Trees with Kripke structures and CTL. Also in this formally founded Attack Tree proof calculus in the present paper, all rules are now proved within Isabelle by virtue of conservative extension and the Correctness and Completeness meta-theorems whose proofs are new mathematical theorems that are additionally performed fully within Isabelle.

2.3. Kripke Structures and CTL

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 security and privacy properties.

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 lifts Kripke structures and CTL to a general level. The definition of the inductive relation is given by a set of specific rules which are, however, part of an application like quantum key distribution (Section 6.2). 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 security protocols.

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

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

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 – an operator supplied by the Isabelle theory library – applied to the relation →.

The `Kripke` constructor combines the constituents initial state, state set and state transition relation →.

```
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 equivalent 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.

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

3. Attack Trees and Refinement

Attack Trees [11] 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 3 is an example of an attack tree taken from a textbook [11] illustrating the attack of opening a safe. Nodes can be adorned with attributes, for example

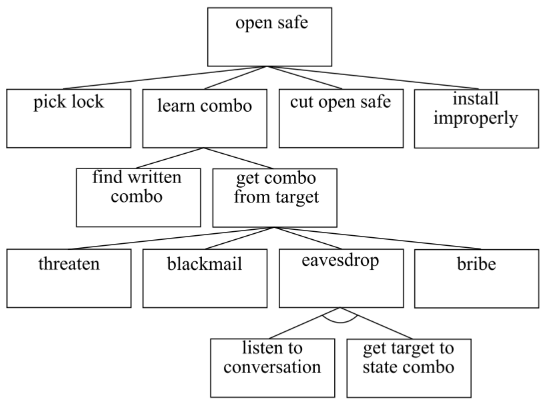

Figure 3: Attack tree example illustrating disjunctive nodes for alternative attacks refining the attack "open safe". Near the leaves there is also a conjunctive node "eavesdrop".

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) ("_ ⊕∨^(_)")
```

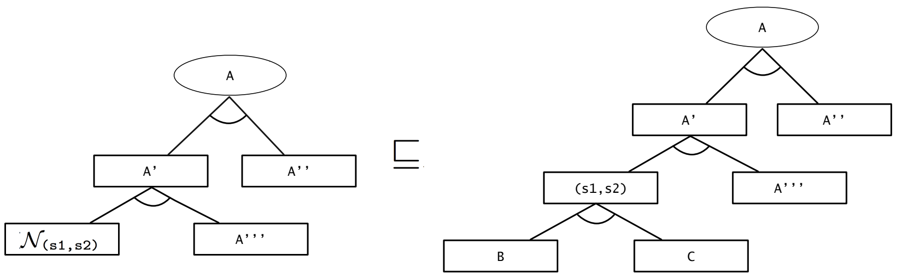

Figure 4: Attack tree example illustrating refinement of an and-subtree.

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`.

```
 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 4. 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 the following inductive definition for the operator `refines_to` with infix syntax $\sqsubseteq$. The inductive definition is given by the smallest predicate closed under the set of specified rules, here `refI, ref_or, ref_trans` and `ref_refl`.

```
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 4: 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` $\oplus_{\wedge}^{\texttt{(s0,s1)}}$ 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 and a necessary finalisation to the inductive definition provided in the preliminary workshop paper [2] that inspired this paper. Our notion of attack tree validity here is now fully rooted on the syntactic definition of attacks using only basic logical operators and lists to define the truth value of a $\vdash$-formula. This allows to infer properties essential for proofs and also for linking in the notion of attack tree refinement as we will see shortly. Consistency is now provided because any other important or useful algebraic property 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 ad hoc properties of attack validity as inductive rules. While inductive definitions cannot introduce inconsistencies into the logic, they may well be inconsistent in themselves. Any attack tree validity properties are now proved from the three cases of `is_attack_tree` definition.

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 definition of the state transition for security protocols in Section 6.2. 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.

$$\texttt{A} \sqsubseteq_V \texttt{A'} \equiv (\texttt{A} \sqsubseteq \texttt{A'} \wedge \vdash \texttt{A'})$$

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.

$$\vdash_V \texttt{A} \equiv (\exists\, \texttt{A'}.\ \texttt{A} \sqsubseteq_V \texttt{A'})$$

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 $\vdash$`A'` such that `A` $\sqsubseteq$ `A'`. For practical purposes, the following lemma implements this method.

lemma `ref_valI`: `A` $\sqsubseteq$ `A'` $\Longrightarrow$ $\vdash$ `A'` $\Longrightarrow$ $\vdash_V$ `A`

We are going to use this method on the case study in Section 6.3 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`: $\vdash$ `A :: (`σ` :: state) attree)` $\Longrightarrow$ `(I, s) = attack A`
$\Longrightarrow$ `Kripke {t .` $\exists$ `i` $\in$ `I. i` $\rightarrow$`^* t} I` $\vdash$ `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 $\vdash$ 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` [4]. The interactive proofs including auxiliary lemmas consist of nearly 1500 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 Example: Quantum Key Distribution

As an application example for Attack Trees, we present a formalisation of parts of the Quantum Key Distribution (QKD) algorithm in Isabelle. In this section, we first briefly introduce the algorithm and then show how the basis for modeling protocols can be built on top of the Attack Tree framework. We then show how Attack Trees and Attack Tree refinement can be used to establish that the attacker Eve can intercept the transmitted key. Clearly, in addition to showing the possibility of the attack, we are interested in quantifying it. This motivates the extension of the Attack Tree framework to probabilistic reasoning and will be presented in the subsequent section.

6.1. Quantum Key Distribution Algorithm

Quantum Key Distribution (QKD) is a security protocol that can be used to transmit a sequence of random bits (that can then be used as a shared One-Time-Pad key giving 100% security)[3]. In each step of the algorithm Alice invents a random bit and sends it to Bob as follows.

(1) Alice randomly selects a bit 0 or 1

(2) Alice randomly chooses diagonal ($\times$) or rectilinear ($+$) polarisation schemes to encode the bit as a photon before sending the bit

(3) Bob also randomly chooses schemes ($\times/+$) before measuring the received photon. According to quantum properties, if Alice and Bob chose the same polarisation schemes the transmission is 100% correct – if they use different ones the chances are 50/50.

[3]An introduction to this protocol for non-physicists that is also largely sufficient for the context of this paper and very entertaining is the popular science book [12]; similar but deeper is [13].

Alice randomly selects a key bit										
Alice's key bit	0	1	0	0	1	0	1	0	1	1
Alice's polarisation scheme	×	+	+	×	+	×	×	+	+	+
Photon sent	↗	↑	→	↗	↑	↗	↖	→	↑	↑
Bob (Eve) choses a polarisation scheme and measures										
Bob(Eve)'s polarisation	×	+	×	+	+	+	×	+	×	×
Bob(Eve) measures	↗	↑	↖	→	↑	↑	↖	→	↖	↗
Received bit	0	1	1	0	1	1	1	0	1	0
They can use those bits where polarisation schemes coincide										
Correctly transmitted	0	1	-	-	1	-	1	0	-	-

Figure 5: QKD examples for key bit selected by Alice, polarisation schemes chosen at sending and receiving end, and which bits can finally be retained (after a final protocol step of comparing the chosen polarisation schemes (omitted here).

Applying polarisation schemes to photons influences the orientation of the photons and has an effect on their measurement at the receiving end. A representative list of possible combinations is given in Figure 5.

Note, that we consider only one bit since the principle is the same in any number of repetitions necessary to transmit a n-bit key. Also we only consider for a start the first phase of the protocol.

6.2. Formalizing Security Protocols with Example QKD

In this section, we show how security protocols can be modeled in Isabelle in a simple way by introducing the formalisation of QKD on top of the Attack Tree framework. The basic idea is to define a set of sequences of events, for example, messages being sent, polarisation schemes chosen, or bits received, representing the state of ongoing communications. For simplicity of the model, these events are introduced as a datatype enumerating Boolean constructors, for example, "Alice sends bit 1" is encoded as `AsOne True`, "Bob choses diagonal polarisation scheme" as `BchX True`, and "Eve measures 1" as `EmOne True`.

```
datatype event = AsOne bool |  AchX bool | BchX bool | EchX bool |
                   BmOne bool | EmOne bool
```

We do not need to encode the events for 0 nor for chosing the rectilinear scheme: for example, "Alice sends bit 0" simply corresponds to `AsOne False` and "B choses the rectilinear scheme" to `BchX False`. The possibilities of the attacker follow the classical model of Dolev-Yao [14] that makes strong assumptions on the attacker: Eve can read all communications, can intercept and feed in new messages based on what she has intercepted and could analyse from that. We define a datatype for protocol states as sets of lists of events by the single constructor `Protocol` that takes an element of type `event list set` as input. We use the polymorphic Isabelle type constructors list and set which – applied as postfix constructors to any type α (here `event list set`)– yield the type of sets or lists over α, respectively.

```
datatype protocol = Protocol event list set
```

The protocol steps are then defined by the inductive definition of the state transition where each step of the protocol is encoded in one rule of this inductive definition.

The rules use the current state, in particular, the sequences of events `evs` that have occurred up to this point, to define how `evs` evolves in one step. For example, the first rule `AsendsBitB` only assumes that the empty set `[]` is in the current state `evs` and allows to conclude that then the initial sequence "A has sent key bit 1" is in the following state by inserting the one element sequence `[AsOne b]` into the next state reachable by the state transition relation $\to_Q$ from the current state `evs`. Note, that the rule uses the variable `b` for some `b` $\in$ {`True, False`} which allows two instances of this rule. The other rules follow the same pattern but additionally use more structured assumptions on some existing sequence `l` present in the current state `evs`. In doing so, they make use of the list constructors `a # l` for putting the element `a` as first element into list `l` and again `hd l` to return the first element and `tl l` to return the tail of list `l`, respectively. We defined the function `insertp` to insert a new `event list` into the `protocol` (which contains an event list set by definition).

```
inductive state_transition_qkd :: [protocol, protocol] ⇒ bool ("_ →Q _")
where
AsendsBitb:  [] ∈ evs ⟹ evs →Q insertp [AsOne b] evs
AchosesPolX: l ∈ evs ⟹ hd l = AsOne b ⟹
             evs →Q insertp (AchX b’ # l) evs
BchosesPolX: l ∈ evs ⟹ hd l = AchX b ⟹
             evs →Q insertp (BchX b’ # l) evs
BmeasuresOK: l ∈ evs ⟹ hd l = BchX b ⟹ hd(tl l) = AchX b ⟹
             hd(tl(tl l)) = AsOne b’ ⟹
             evs →Q insertp (BmOne b’ # l) evs
BmeasureNOK: l ∈ evs ⟹ hd l = BchX b ⟹ hd(tl l)) = AchX b’ ⟹
             b ≠ b’ ⟹
             evs →Q insertp (BmOne b’’ # l) evs
EchosesPolX: l ∈ evs ⟹ hd l = AchX b ⟹
             evs →Q insertp (EchX b’ # l) evs
EintercptOK: l ∈ evs ⟹ hd l = Ech b ⟹ hd(tl l) = AchX b ⟹
             hd(tl(tl l)) = AsOne b’ ⟹
             evs →Q insertp (EmOne b’ # l) evs
EintrcptNOK: l ∈ evs ⟹ hd l = Ech b ⟹ hd(tl l)) = AchX b’ ⟹
             b ≠ b’ ⟹
             evs →Q insertp (EmOne b’’ # l) evs
B_E_iterate: l ∈ evs ⟹ hd l = BmOne b ∨ hd l = EmOne b ⟹
             evs →Q insertp (AsOne b # l) evs
```

Note that in the rule `BmeasuresOK` – and similar for `EintercptOK` – the same variable `b` is used in `BchX b` and `AchX b`: thereby we presume that `A` and `B` have chosen the same polarisation filter. We can conclude that then `B` measures correctly. By contrast, in rule `BmeasuresNOK` – and similar for the corresponding rule `EintrcptNOK` – this is not the case if `b` $\neq$ `b’`: then one of them is `True` and the other is False. This means A and B use different polarisation filters and hence the measurement outcome is unclear. This is encoded in the rule by a new variable b'' which could be equal to either True or False.

The last rule B_E_iterate is very similar to AsendsBitb allowing the start of a follow up run of the protocol (to send the next key bit). The inductive definition defines the smallest state transition relation that is closed under applying those rules. In particular, this allows also states representing execution of various parallel runs of the protocol. This is not relevant for QKD but in general for the analysis of other protocols in particular to detect man-in-the-middle attacks.

6.3. Attack Tree Analysis of QKD

We define the global policy as "Eve must not learn the key" where the key is the transmitted bit. To this end, we first need to specify the set of event sequences in which Alice sent a key bit and Eve measured this key bit correctly. In other words, we need to characterize such event lists in which A sent bit b and Eve measured the same bit b. The actual choices of the orientation of the polarisation filters by Alice and Eve and whether they match does not matter – even if the filter choices mismatch there could coincidentally be a matching outcome (see column 4 of Figure 5). The two relevant cases are then negated using ¬ since we do not want this to happen. Since the event sequences are lists we apply the function set to transform them into finite sets so that we can simply use the set operations ∈ to specify membership. The predicate global_policy specifies this for an arbitrary state e::event set. Note that we can generalise the two relevant cases for True and False by ∀ b where b::bool.

```
global_policy e ≡ ∀ l ∈ e. ∀ b. ¬(AsOne b ∈ set l ∧ EmOne b ∈ set l)
```

How do we find attacks? The key is to use invalidation [15] of the security property we want to achieve, here the global policy. Since we consider a predicate transformer semantics, we use sets of states to represent properties. The attack tree is given by starting from this invalidated global policy and apply attack refinement from there. To define the attack property that represents the root of the attack tree refinement, we need to logically characterize all states in which the global policy is not valid. The attack property is given by the following set negated_policy.

```
negated_policy ≡ { e :: protocol. ¬ global_policy e }
```

We first define an initial state as the set of event lists only containing the empty list which is the starting point for any protocol execution.

```
qkd_scenario ≡ Protocol { ([]:: event list) }
```

Using this scenario to build a simple initial state set Iqkd ≡{qkd_scenario}, we define a Kripke structure.

```
  qkd_Kripke ≡ Kripke { I. qkd_scenario →*_Q I } Iqkd
```

The attack we are interested in is to see whether the critical state `negated_policy` can be reached, i.e., is there a valid attack `(Iqkd,negated_policy)`?

The states $\texttt{QKD}_i$ for $i \in \{1..3\}$ are intermediate states of protocol executions that we will use to establish the attack.

```
QKD1 ≡ insertp [AsOne True] qkd_scenario
QKD2 ≡ insertp [AchX True, AsOne True] QKD1
QKD3 ≡ insertp [EchX True, AchX True, AsOne True] QKD2
```

This sequence of states can be systematically derived using the attack tree refinement. Step by step proof establishes that the intermediate steps $\texttt{QKD}_i$ form the following refinement.

$$[\mathcal{N}_{\texttt{(Iqkd,negated\_policy)}}] \oplus_{\wedge}^{\texttt{(Iqkd,negated\_policy)}}$$
$$\sqsubseteq$$
$$[\mathcal{N}_{\texttt{(Iqkd,QKD1)}}, \mathcal{N}_{\texttt{(QKD1,QKD2)}}, \mathcal{N}_{\texttt{(QKD3,negated\_policy)}}] \oplus_{\wedge}^{\texttt{(Iqkd,negate\_policy)}}$$

For the Kripke structure `qkd_Kripke`, we can derive that the refined attack is a valid and-attack using the attack tree proof calculus.

$$\vdash \; [\mathcal{N}_{\texttt{(Iqkd,QKD1)}}, \mathcal{N}_{\texttt{(QKD1,QKD2)}}, \mathcal{N}_{\texttt{(QKD3,negated\_policy)}}] \oplus_{\wedge}^{\texttt{(Iqkd,negated\_policy)}}$$

Application of Theorem `ref_valI` from Section 4 then immediately proves that the (abstract) attack is valid.

$$\vdash_V \; [\mathcal{N}_{\texttt{(Iqkd,negated\_policy)}}] \oplus_{\wedge}^{\texttt{(Iqkd,negated\_policy)}}$$

We can now simply apply the Correctness theorem `AT_EF` to immediately prove the following CTL statement.

```
qkd_Kripke ⊢  EF negated_policy
```

This application of the meta-theorem of Correctness of attack trees saves us proving the CTL formula tediously by exploring the state space.

Note that the states $\texttt{QKD}_i$, $i \in \{1..3\}$ represent just one possible sequence of protocol runs that lead to the attack property `negated_policy`. One immediately obvious other candidate is the variation with `False` but also the following sequence may lead into the attack property.

```
QKDa1 ≡ insertp [AsOne True] qkd_scenario
QKDa2 ≡ insertp [AchX False, AsOne True] QKDa1
QKDa3 ≡ insertp [EchX True, AchX False, AsOne True] QKDa2
```

Although Eve choses the wrong polarisation scheme, she might still be successful. We can establish this by proving that this is a valid attack, i.e., may lead to `negated_policy` but it is not fully satisfactory. What we really want is to understand the likelihood of even such seemingly unlikely attacks. This leads on to a decisive extension of the framework to probabilistic attack analysis.

7. Probabilistic Attack Analysis

In this section, we introduce the necessary basic probability theory to lay a foundation to extend the temporal logic CTL to Probabilistic CTL (PCTL). We then use this extension to illustrate how it can be employed for probabilistic attack analysis to prove quantitative properties of QKD relevant for security analysis.

7.1. Brief Excursion into Probability

We develop and illustrate the probability reasoning on finite sets of outcomes in Isabelle. The very brief introduction to basic probability theory is taken from Koller and Friedmann [16] but vastly abbreviated. The reader is referred to this excellent textbook for details.

Before defining events $(\mathcal{S})$[4] we first assume a set Ω of possible outcomes. Based on that we define a set of *measurable* events $\mathcal{S} \subseteq \mathcal{P}\Omega$ where $\mathcal{P}$ is the power set, that is, the set of subsets of a set Ω. Any event $A \in \mathcal{S}$ may have probabilities assigned to it. Probability theory, more precisely, measure theory (see [17]), requires that the following conditions hold for the probability space $\mathcal{S}$:

- $\mathcal{S}$ contains the empty event $\varnothing$ and the trivial event Ω;
- $\mathcal{S}$ must be closed under union: $A, B \in \mathcal{S} \Rightarrow A \cup B \in \mathcal{S}$;
- $\mathcal{S}$ must be closed under complement: $A \in \mathcal{S} \Rightarrow \Omega \setminus A \in \mathcal{S}$.

The closure for the other Boolean operators intersection and set difference is implied by the above conditions.

Definition 7.1 (Probability Distribution)**.** *A probability distribution P over $(\Omega, \mathcal{S})$ is a function from events in $\mathcal{S}$ to real numbers satisfying the following conditions.*

1. $\forall A \in \mathcal{S}.\ P(A) \geq 0.$
2. $P(\Omega) = 1.$
3. *If* $A, B \in \mathcal{S}$ *and* $A \cap B = \varnothing$ *then* $P(A \cup B) = P(A) + P(B)$.

In Joe Hurd's dissertation [17] these conditions are referred to as

(1) *Positivity*,

(2) *Probability space* ((2.27), page 33 [17]), and

(3) *Additivity*

[4]Note, that confusingly probability theory also uses the terminology "event" in a slightly different way than it is common in security protocols as seen in the previous section.

in the general context of Measure spaces. The property of Monotonicity and Countable Additivity [17] are not present in the introduction of Koller and Friedman but at least Countable Additivity can be considered as implicit since we are looking at finite spaces only.

The above definition can be directly translated into an Isabelle specification[5]. We transform the textbook definition into a definition and a type definition: we define first event spaces over finite types of outcomes and then we give a type definition for probability distribution.

The possible outcomes can be provided as a type represented here by a type variable Ω. This type is assumed to be finite implicitly by coercing the type variable Ω into the type class `finite` using the type judgment with `::` in the following definition of probability space.

```
definition prob_space :: ((Ω :: finite) set) set) ⇒ bool
where prob_space 𝒮 ≡ {} ∈ 𝒮 ∧ (UNIV :: Ω set) ∈ 𝒮 ∧
                     ∀ A, B ∈ 𝒮. A ∪ B ∈ 𝒮 ∧
                     ∀ A ∈ 𝒮. (UNIV :: Ω set) - A ∈ 𝒮
```

In the above type definition, the Ω is an Isabelle type variable. The polymorphic constructor `UNIV` is a standard constructor in Isabelle and represents the set of all elements of a type, here all outcomes in Ω. We can now show that the power set over a finite type is a probability space.

```
theorem Pow_prob_space: prob_space (Pow (UNIV :: (Ω :: finite) set))
```

A probability distribution is a function over a probability space. We use a type definition for it.

```
typedef (Ω :: finite) prob_dist ≡ {p :: (Ω set ⇒ real).
           ∀ (A :: Ω set). p A ≥ 0 ∧ p(UNIV :: Ω set) = 1 ∧
           ∀ (A :: Ω set) B . A ∩ B = ∅ ⟶ p(A ∪ B) = p(A) + p(B) }
```

In the above type definition for probability distribution, we can see that the three criteria from Definition 7.1 are almost one to one translated into Isabelle. Type definitions are applied by imposing them on new constants or variables which automatically leads to the invocation of the defining properties on these elements: either by assuming them for constants defined over the new types or by creating new proof obligations when existing terms are judged to be of these types. We apply this when we define a probability distribution over the power set of a finite type of outcomes for QKD in Section 7.3.

Hurd already writes "Measure theory defines what probability spaces are but does little to help us find concrete distributions" [17]. He then uses Caratheodory's extension theorem to help out. For the simple case of finite sets of outcomes that we consider here, we introduce a canonical construction that uses the power

[5]Even though measure theory à la Hurd is provided in the Isabelle theory library, we prefer to provide a simpler ad hoc definition here for completeness – integration is possible and planned for later stages.

set of outcomes as the event space and accordingly constructs the probability distribution by summing up the probabilities for the individual outcomes of any subset of Ω, i.e. an event $\in \mathcal{S}$, which is possible since they are finite sets of outcomes that are all distinct. For the definition of a generic operator for this canonical construction, we use the `fold` operator available in Isabelle for defining simple recursive functions over finite sets. Intuitively, `fold` operates like this:

$$\texttt{fold}\ f\ z\ \{x_1, \ldots, x_n\} = f(x_1 \ldots (f\ x_n\ z))\,.$$

We define the canonical construction for probability distributions as a function `pmap` lifting a probability assignment `ops` for single outcomes $\in \Omega$ to any set `S` of outcomes.

```
pmap (ops :: Ω ⇒ real) S = fold (λ x y. ops x + y) 0 S
```

Now, we can show that, for any finite type Ω :: `finite` with a probability assignment `ops`, that is positive and additive to UNIV the canonical construction `pmap ops` yields a probability distribution over the power set. We just need to show that it is contained in the defining set of the type `prob_dist` given by the set `prob_dist_def` that defines the type.

```
theorem pmap_ops: ∀ x :: (Ω :: finite). 0 ≤ ops x  ⟹
               Σ ops (UNIV :: Ω set) = 1 ⟹
                pmap ops ∈ prob_dist_def
```

Conditional probability, for example, $P(A|B)$ signifies the probability for an event A given an event B. It can be defined simply as follows.

Definition 7.2 (Conditional Probability)**.** *For an event space $\mathcal{S}$ and two events $A, B \in \mathcal{S}$ the conditional probability of A given B is defined for a probability distribution P as*

$$P(A|B) \equiv P(A \cap B)/P(B)\,.$$

The corresponding Isabelle definition uses some syntactic sugaring to enable a one-to-one notation of the conditional probability operator. Note, the actual operator has three inputs: one probability distribution `P` and two events `A, B`.

```
definition cond_prob :: (Ω :: finite)prob_dist ⇒ Ω set ⇒ Ω set ⇒ real
                                                              ("_(_|_)")
where   P(A|B) ≡ (Rep_prob_dist P (A ∩ B)) / (Rep_prob_dist P B)
```

The above Isabelle definition uses the mixfix syntax after the type in quotation marks to allow writing the same probability distribution as a function with two arguments by a syntactic translation into the corresponding definition with intersections of the event sets.

7.2. Probabilistic CTL

To extend the temporal logic CTL to a probabilistic version, we follow the theoretical description in the texbook [18] but adapt it to the finite lists we consider here for security protocols and QKD.

The main idea of our extension is to add a new operator to express the probability of sequences of state changes. In other word, we define a path quantifier that selects a number of state transition sequences in a Kripke structure and then uses the probability distribution to compute the probability for those paths to occur. This will enable quantifying the likelihood of certain attack paths to be reached. So, the outcomes that are quantified with probabilities by the probability distribution will be paths leading into attack states. We introduce a "finally" (F) path quantifier. This definition uses two operators from Isabelle's rich theory library on lists: the operator `nth` written as an infix `!` returns the "nth" list element, for example, `[a,b,c] ! 2 = c`; the operator `last` returns the "last", i.e., rightmost, element of a list, for example, `last[1,2,3] = 3`.

```
definition F :: (α :: state) set ⇒ α list set
where F s = { l. ∀ i < length l. l ! i → l ! (Suc i) ∧ last l ∈ s }
```

We usually want this path quantifier to be relative to a Kripke structure `M` because we want the attack paths to start from initial states and be paths along states within `M`. To this end, we define the following specialised version relative to a Kripke model `M` (the operator `hd` yields the head (first element) of a list).

```
definition ⊢F :: [(α :: state) Kripke, α set] ⇒ α list set
M ⊢F s ≡ { l. set l ⊆ states M ∧ hd l ∈ init M } ∩ F s
```

Given the operator `F` to select a set of finite paths leading into a state, we can now combine this with the probability notions of the previous section. For example, to introduce the probability for a set of finite paths given an assignment `ops` of probabilities to paths representing outcomes of protocol runs, then `pmap ops` produces a probability distribution for this protocol. This can then be applied to the previous path selection. That is, `pmap ops (M ⊢Fs)` computes the probability of the event that paths end in state `s`.

This probability computation leads finally into a logical operator by considering predicates J over real numbers. For example, $J{\equiv}(\lambda x.x = 1)$ is a predicate that is only true for 1 and $\lambda x.0 < x \leq 1/2$ is true for all positive real values less than one half. We also need to relate the path quantification with a Kripke structure. This comes together in the logic PCTL.

Definition 7.3 (PCTL). *PCTL extends CTL by a probabilistic operator* PF_J *where* ϕ *is a set of finite paths and* `J` *is a predicate defining an interval of* $[0, 1]$. *Formally,*

```
definition probF::[α Kripke, α list set ⇒ real,
                   real ⇒ bool, α set]⇒ bool            ("_ _ ⊢PF_ _")
M pdist ⊢PF_J s ≡ J(pdist (M ⊢F s))
```

The predicate `J` indicates a lower bound and/or upper bound on the probability. Since `pmap ops (M ⊢F s)` yields the probability of paths ending in `s`, the intuitive meaning of the formula $\mathsf{PF}_\mathtt{J}$ is: the probability for the set of paths satisfying the bounds given by `J`. The probabilistic operator can be considered as the quantitative counterpart to the CTL path quantifiers A and E.

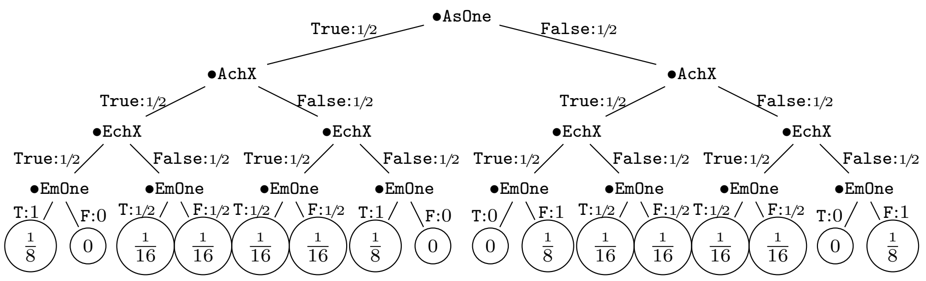

Figure 6: Outcome tree for QKD: probabilities for each step at edges and outcome probabilities as leaves.

7.3. QKD Probability Model

The tree depicted in Figure 7.3 is not an attack tree: it is a simple depiction that shows the probabilities along the paths from root to leaves that correspond to basic outcomes of protocol runs in which Eve intercepts.

The basic distribution on these 16 outcomes derived from Figure 7.3 is given in the table in Figure 8. To facilitate the definition of the probability distribution, we define a finite type of outcomes specifying event lists of length four.

```
typedef outcome = { l :: event list. length l = 4 }
```

To define the basic outcome function `qkd_ops`, we use a locale definition [19].

```
 defines (qkd_ops  :: outcome ⇒ real) =
      λ x. case Rep_outcome x of
            [EmOne False, EchX False, AchX False, AsOne False] ⇒ 1/8
          | [EmOne True, EchX False, AchX False, AsOne False] ⇒ 0
          | [EmOne False, EchX True, AchX False, AsOne False] ⇒ 1/16
          | [EmOne True, EchX True, AchX False, AsOne False] ⇒ 1/16
          | [EmOne False, EchX False, AchX True, AsOne False] ⇒ 1/16
          | [EmOne True, EchX False, AchX True, AsOne False] ⇒ 1/16
          | [EmOne False, EchX True, AchX True, AsOne False] ⇒ 1/8
          | [EmOne True, EchX True, AchX True, AsOne False] ⇒ 0
          | [EmOne False, EchX False, AchX False, AsOne True] ⇒ 0
          | [EmOne True, EchX False, AchX False, AsOne True] ⇒ 1/8
          | [EmOne False, EchX True, AchX False, AsOne True] ⇒ 1/16
          | [EmOne True, EchX True, AchX False, AsOne True] ⇒ 1/16
          | [EmOne False, EchX False, AchX True, AsOne True] ⇒ 1/16
          | [EmOne True, EchX False, AchX True, AsOne True] ⇒ 1/16
          | [EmOne False, EchX True, AchX True, AsOne True] ⇒ 0
          | [EmOne True, EchX True, AchX True, AsOne True] ⇒ 1/8
          |  _ ⇒ 0
```

The probability distribution `qkd_ops` can be input into the above defined function `pmap` producing automatically the canonical probability distribution for the QKD protocol. We thus define a probability distribution being able to show that

AsOne	AchX	EchX	EmOne	P
False	False	False	False	1/8
False	False	False	True	0
False	False	True	False	1/16
False	False	True	True	1/16
False	True	False	False	1/16
False	True	False	True	1/16
False	True	True	False	1/8
False	True	True	True	0
True	False	False	False	0
True	False	False	True	1/8
True	False	True	False	1/16
True	False	True	True	1/16
True	True	False	False	1/16
True	True	False	True	1/16
True	True	True	False	0
True	True	True	True	1/8

Figure 7: Probability assignment for QKD outcomes

it is in fact one in the following lemma that shows that `pmap` maps the assignment of outcomes to probabilities given by `qkd_ops` into the set `prob_dist_def` that defines probability distributions over finite types.

lemma `qkd_prob_dist: pmap qkd_ops` $\in$ `prob_dist_def`

The proof of this lemma applies the general theorem `pmap_ops` we proved earlier. Based on this probability distribution we can calculate interesting probabilities telling us – in a mathematical precise way – something about the security of the protocol.

In order to do that, we first consider another useful probability law: the law of total probability.

Theorem 1 (Law of total probability)**.** *Let $A_j, j \leq n$ for some $n \in \mathbb{N}$ be a set of events partitioning the event space $\mathcal{S}$, that is, $\forall\ i, j \leq n.\ i \neq j \Rightarrow A_i \cap A_j = \varnothing$ and $\bigcup_j A_j = \Omega$. Let further $B \in \mathcal{S}$. We then have that*

$$P(B) = \sum_j P(B|A_j)P(A_j)\,.$$

Proof: Since we have a partition, that is, $A_i \cap A_j = \varnothing$ for all $i, j \leq n$ with $i \neq j$, we have also

$$(B \cap A_i) \cap (B \cap A_j) \quad = \quad B \cap (A_i \cap A_j) \quad = \quad B \cap \varnothing \quad = \quad \varnothing \tag{1}$$

Therefore

$$\begin{array}{rcll} P(B) & = & P(B \cap \Omega) & (A_j \text{ is partition of } \Omega) \\ & = & P(B \cap (A_1 \cup \cdots \cup A_n)) & \text{(set algebra)} \\ & = & P((B \cap A_1) \cup \cdots \cup (B \cap A_n)) & \text{((1) and Definition 7.1 (3))} \\ & = & P(B \cap A_1) + \cdots + P(B \cap A_n) & \text{(summation)} \\ & = & \sum_j P(B \cap A_j) & \text{(Def. 7.2 Conditional Probability)} \\ & = & \sum_j P(B|A_j) * P(A_j) & \square \end{array}$$

7.4. Probabilistic Attack Analysis

The first security argument of the attack analysis computes the probability that E measures 1 applying the law of total probability. The partition $\mathcal{A}$ of Ω used in the derivation is given as the following family of disjoint sets A_j with $\mathcal{A} = \bigcup_{j \in \{0..7\}} A_j = \Omega =$ `UNIV::outcome set`.

$$\begin{array}{rcl} \mathcal{A} & = & \{\{s :: \texttt{outcome}.\ \exists \texttt{e}.\ s = \texttt{[e, EchX True, AchX True, AsOne True]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX False, AchX True, AsOne True]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX True, AchX False, AsOne True]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX False, AchX False, AsOne True]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX True, AchX True, AsOne False]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX False, AchX True, AsOne False]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX True, AchX False, AsOne False]}\}, \\ & & \{s.\ \exists \texttt{e}.\ s = \texttt{[e, EchX False, AchX False, AsOne False]}\}, \\ & & \} \end{array}$$

For each $A_j \in \mathcal{A}$, we have $P(A_j) = 1/8$: since P is a probability distribution, we can use the third defining property of the type definition `prob_dest` to sum up the disjoint probabilities for each outcome. The outcome probabilities in Figure 8 give for example for A_0 (similar for the other A_j):

$$\begin{array}{ll} P(\{s :: \texttt{outcome}.\ \exists \texttt{e}.\ s = \texttt{[e, EchX True, AchX True, AsOne True]}\}) & = \\ P(\texttt{[EmOne True, EchX True, AchX True, AsOne True]}) \ + & \\ P(\texttt{[EmOne False, EchX True, AchX True, AsOne True]}) & = \\ 1/8 + 0 = 1/8 & \end{array}$$

We are interested in calculating the probability of Eve being able to receive a key bit, say, one which we define as the event `EmOne'`.

definition `EmOne' :: outcome set ≡`
```
                { l. hd (Rep_outcome l) = ((EmOne True) :: event) }
```

With this we can compute that $P(\texttt{EmOne'}) = 1/2$.

$$\begin{array}{rcll} P(\texttt{EmOne'}) & = & \sum_{A_j \in \mathcal{A}} P(\texttt{EmOne'}|A_j) * P(A_j) & \text{(Theorem 1)} \\ & = & 1/8 * \sum_{A_j \in \mathcal{A}} P(\texttt{EmOne'}|A_j) & (P(A_j) = 1/8) \\ & = & 1/8 * \sum_{A_j \in \mathcal{A}} P(\texttt{EmOne'} \cap A_j)/P(A_j) & \text{(Definition 7.2)} \\ & = & \sum_{A_j \in \mathcal{A}} P(\texttt{EmOne'} \cap A_j) & (P(A_j) = 1/8) \\ & = & 1/2 & \text{(sum, see Table 8)} \end{array}$$

This probability cannot be interpreted as a security statement directly. It rather says that on the whole Eve may receive 1s with 50% probability but not how this relates to what A has actually sent. However, the above probability $P(\texttt{EmOne'})$ is useful to calculate the conditional probability $P(\texttt{AsOne'}|\texttt{EmOne'})$: how likely is it that A has actually sent a 1 given that E intercepted a 1? Similar to the above event `EmOne'`, we define here event `AsOne'` as $\{$`l.l!3 = AsOne True`$\}$. For the sake of exposition, we abbreviate the outcomes, for example, we write `[F,F,T,T]` for the outcome `[EmOne False, EchX False, AchX True, AsOne True]`.

$$\begin{array}{rcll} P(\texttt{AsOne'}|\texttt{EmOne'}) & = & P(\texttt{AsOne'} \cap \texttt{EmOne'})/P(\texttt{EmOne'}) & \text{(Def. 7.2)} \\ & = & 2 * (P(\texttt{AsOne'} \;\cap \texttt{EmOne'})) & \text{(above)} \\ & = & 2 * (P(\{\texttt{[T,F,F,T]}, \texttt{[T,T,F,T]}, & \\ & & \quad \texttt{[T,F,T,T]}, \texttt{[T,T,T,T]}\})) & \text{(Def. 7.1.3)} \\ & = & 2 * (P(\texttt{[T,F,F,T]}) + P(\texttt{[T,T,F,T]}) + & \\ & & \quad P(\texttt{[T,F,T,T]}) + P(\texttt{[T,T,T,T]})) & \text{(Table 8)} \\ & = & 2 * (1/8 + 1/16 + 1/16 + 1/8) & \text{(arithmetic)} \\ & = & 3/4 & \end{array}$$

This shows that there is 25% chance of error for Eve to receive the wrong bit.

PCTL serves to represent this mathematical calculation as a logical statement. From the above, we can directly prove the following PCTL statement.

lemma `qkd_Eve_attack:`
  `qkd_Kripke (pmap qkd_ops)` $\vdash \mathsf{PF}_{\lambda \texttt{ x. x = 3/4}}$ `negated_policy`

Thereby, we have nicely wrapped up probability theory and temporal logic in our Isabelle framework and showed that it enables quantifying temporal attack properties.

Moreover, we can now additionally employ the completeness of attack trees. Since we have shown the CTL statement (see Section 6.3)

```
qkd_Kripke ⊢  EF negated_policy
```

we can apply Theorem Completeness (see Section 5) and thereby prove

```
∃ A :: ((event list set)::state)attree.
     ⊢ A ∧ (I, s) = (Iqkd,negated_policy)
```

but now additionally quantifying this attack using PCTL by lemma `qkd_Eve_attack` to have the probability 75%.

8. Conclusions

8.1. Summary

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 $\mathsf{EF}\ s$ formulas. Practical relevance is supported by a clear and generic structure of the framework with classes for arbitrary instantiation and generated code for automation of verification.

Extending this foundation of attack trees, already presented in [3], we have first illustrated its application to security protocols focussing on the Quantum Key Distribution protocol (QKD). We have shown how protocols can be simply expressed in the generic attack tree framework and how this can then be used to formalise the QKD protocol. Attack tree refinement can be used to establish the attack where Eve can get the secret key bit. However, the QKD case study also shows the need for a quantitative analysis. We thus introduced probabilities extending the Isabelle framework by basic concepts. We then used this foundation to define Probabilistic CTL (PCTL) enabling proof of quantified statements. We showed how this can finally be applied to provide probabilistic statements for attacks illustrating it on the QKD protocol.

8.2. Related Work

There are excellent foundations for attack trees available based on graph theory [20]. They provide a very good understanding of the formalism, various extensions (like attack-defence trees [21]) and differentiations of the operators (like sequential conjunction (SAND) versus parallel conjunction [22]) 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 semantic 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 [21]. The theoretical foundations allow comparison of different semantics, and provide a theoretical framework to develop evaluation algorithms for the quantification of attacks. Other verification approaches based on Modelchecking, e.g. [23], 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.

However, all of these approaches represent attacks merely as action sequences omitting other relevant information of system states, for example, data related information. A notable exception that uses, like our approach, a state based semantics for attack trees is the recent work [24]. However, this work is aiming at assisted generation of attack trees from system models not on verification of secure systems or protocols. The tool ATSyRA supports this process. The paper [24] 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 proven 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.

Compared to other verification techniques, like Modelchecking, Isabelle requires user interaction. However, Modelchecking is restricted to finite models and first order logic. The relationship between Higher Order logic and Modelchecking has been first explored by Kobayashi (see [25] for a paper subsuming previous results). Modelchecking has been realized as well in Isabelle [26] but we formalise the different logic of CTL [27] (instead of LTL) and extend it to the probabilistic logic PCTL.

Isabelle enables the use of higher order quantification and induction necessary for invariant proofs. Powerful concepts like recursive functions in HOL and simplification and other proof tactics in Isabelle furthermore facilitate the application. The principle of conservative extension is key to guaranteeing consistency between the theoretical foundations (we referred to this as the meta-theory) and the application logic (for example, protocols). Moreover, for executable parts of the theory, code can be generated into programming languages like Scala to outsource some of the proof obligations to external fully automated processes – like the central validity checker for attacks (available on GitHub [4]) generated from our Isabelle definition.

Formalising Quantum Cryptography is only an application example illustrating the merits of the Attack Tree framework and motivating the extension to PCTL yet it is worth considering related approaches. Using interactive theorem proving to formalise Quantum Cryptography has been attempted in a deep embedding of a mathematical model for quantum computation and measurement in Coq [28] but without probabilities nor considering QKD. The way of formalising protocols in the current paper is vaguely inspired by Paulson's inductive approach [29] but much simplified in the basics yet surpassing it in using probabilities and addressing Quantum Cryptography and analysing QKD.

We have shown that Isabelle's Higher Order Logic is capable of expressing attack trees in a semantically founded way such that state transitions over protocol executions can be analysed using alternatively CTL or attack refinement. The genericity of the attack tree formalisation presented here permitted the extension of the CTL temporal logic to PCTL enabling probabilistic reasoning which has been illustrated on the Quantum Key Distribution protocol.

References

[1] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer-Verlag, 2002.

[2] F. Kammüller, A proof calculus for attack trees, in: Data Privacy Management, DPM'17, 12th Int. Workshop, Vol. 10436 of LNCS, Springer, 2017, co-located with ESORICS'17.

[3] F. Kammüller, Attack trees in isabelle, in: 20th International Conference on Information and Communications Security, ICICS2018, Vol. 11149 of LNCS, Springer, 2018.

[4] F. Kammüller, Isabelle infrastructure framework with iot healthcare s&p application, available at `https://github.com/flokam/IsabelleAT`. (2018).

[5] F. Kammüller, C. W. Probst, Modeling and verification of insider threats using logical analysis, IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence 11 (2) (2017) 534–545. `doi:10.1109/JSYST.2015.2453215`.
URL `http://dx.doi.org/10.1109/JSYST.2015.2453215`

[6] D. M. Cappelli, A. P. Moore, R. F. Trzeciak, The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud), 1st Edition, SEI Series in Software Engineering, Addison-Wesley Professional, 2012.
URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321812573`

[7] F. Kammüller, M. Kerber, Investigating airplane safety and security against insider threats using logical modeling, in: IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16, IEEE, 2016.

[8] F. Kammüller, M. Kerber, C. Probst, Towards formal analysis of insider threats for auctions, in: 8th ACM CCS International Workshop on Managing Insider Security Threats, MIST'16, ACM, 2016.

[9] F. Kammüller, J. R. C. Nurse, C. W. Probst, Attack tree analysis for insider threats on the IoT using Isabelle, in: Human Aspects of Information Security, Privacy, and Trust - Fourth International Conference, HAS 2015, Held as Part of HCI International 2016, Toronto, Lecture Notes in Computer Science, Springer, 2016, invited paper.

[10] F. Kammüller, Formal modeling and analysis of data protection for gdpr compliance of iot healthcare systems, in: IEEE Systems, Man and Cybernetics, SMC2018, IEEE, 2018.

[11] B. Schneier, Secrets and Lies: Digital Security in a Networked World, John Wiley & Sons, 2004.

[12] S. Singh, The Code Book, 4th Edition, Fourth Estate, 1999.

[13] E. G. Rieffel, W. Polak, An introduction to quantum computing for non-physicists, ACM Comput. Surv. 32 (3) (2000) 300–335.

[14] D. Dolev, A. C. Yao, On the security of public key protocols, in: 22nd Annual Symposium on Foundations of Computer Science, SFCS '81, IEEE, 1981.

[15] F. Kammüller, C. W. Probst, Combining generated data models with formal invalidation for insider threat analysis, in: IEEE Security and Privacy Workshops (SPW), IEEE, 2014.

[16] D. Koller, N. Friedman, Probabilistic Graphical Models – Principles and Techniques, The MIT Press, 2009.

[17] J. Hurd, Formal verification of probabilistic algorithms, Tech. Rep. 566, University of Cambridge (2001).

[18] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.

[19] F. Kammüller, Modular reasoning in isabelle, in: D. MacAllester (Ed.), 17th International Conference on Automated Deduction, CADE-17, Vol. 1831 of LNAI, Springer, 2000.

[20] B. Kordy, L. Piètre-Cambacédés, P. Schweitzer, Dag-based attack and defense modeling: Don't miss the forest for the attack trees, Computer Science Review 13–14 (2014) 1–38.

[21] B. Kordy, S. Mauw, S. Radomirovic, P. Schweitzer, Attack-defense trees, Journal of Logic and Computation 24 (1) (2014) 55–87.

[22] R. Jhawar, B. Kordy, S. Mauw, S. Radomirovic, R. Trujillo-Rasua, Attack trees with sequential conjunction, in: 30th IFIP TC 11 International Conference on ICT Systems Security and Privacy Protection (IFIP SEC'15), Vol. 455 of IFIP Advances in Information and Communication Technology, Springer, 2015, pp. 339–353.

[23] Z. Aslanyan, F. Nielson, D. Parker, Quantitative verification and synthesis of attack-defence scenarios, in: 29th IEEE Computer Security Foundations Symposium, CSF'16, 2016.

[24] M. Audinot, S. Pinchinat, B. Kordy, Is my attack tree correct?, in: 22nd European Symposium on Research in Computer Security, ESORICS'2017, Vol. 10492 of LNCS, Springer, 2017, pp. 83–102.

[25] N. Kobayashi, Model checking higher-order programs, J. ACM 60 (3) (2013) 20:1–20:62. `doi:10.1145/2487241.2487246`.
URL `https://doi.org/10.1145/2487241.2487246`

[26] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, J. Smaus, A fully verified executable LTL model checker, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Vol. 8044 of Lecture Notes in Computer Science, Springer, 2013, pp. 463–478. `doi:10.1007/978-3-642-39799-8\_31`.
URL `https://doi.org/10.1007/978-3-642-39799-8_31`

[27] F. Kammüller, Isabelle modelchecking for insider threats, in: Data Privacy Management, DPM'16, 11th Int. Workshop, Vol. 9963 of LNCS, Springer, 2016, co-located with ESORICS'16.

[28] R. N. J. Boender, F. Kammüller, Formalization of quantum protocols using coq, EPTCS 195.
URL `http://dx.doi.org/10.4204/EPTCS.195`

[29] L. C. Paulson, The inductive approach to verifying cryptographic protocols, Journal of Computer Security 6 (1-2) (1998) 85–128.