2014 IEEE 27th Computer Security Foundations Symposium

# Automated Generation of Attack Trees

Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson
Department of Applied Mathematics and Computer Science
Technical University of Denmark
{rvig,fnie,hrni}@dtu.dk

***Abstract*—Attack trees are widely used to represent threat scenarios in a succinct and intuitive manner, suitable for conveying security information to non-experts. The manual construction of such objects relies on the creativity and experience of specialists, and therefore it is error-prone and impracticable for large systems. Nonetheless, the automated generation of attack trees has only been explored in connection to computer networks and levering rich models, whose analysis typically leads to an exponential blow-up of the state space.**

**We propose a static analysis approach where attack trees are automatically inferred from a process algebraic specification in a syntax-directed fashion, encompassing a great many application domains and avoiding incurring systematically an exponential explosion. Moreover, we show how the standard propositional denotation of an attack tree can be used to phrase interesting quantitative problems, that can be solved through an encoding into Satisfiability Modulo Theories. The flexibility and effectiveness of the approach is demonstrated on the study of a national-scale authentication system, whose attack tree is computed thanks to a Java implementation of the framework.**

## I. Introduction

Physical, software, and cyber-physical systems govern our everyday life increasingly, and are exploited in the realisation of critical infrastructure, whose security is a public concern. The growing complexity of such systems demand for a thorough investigation of the attack scenarios that threaten their operation, and for a quantitative evaluation of their likelihood and criticality.

Attack trees are a widely-used graphical formalism for representing threat scenarios, as they appeal both to scientists, for it is possible to assign them a formal semantics, and to practitioners, for they convey their message in a concise and intuitive way. In an attack tree, the root represents a target goal, while the leaves contain basic attacks whose further refinement is impossible or can be neglected. Internal nodes show how the sub-trees have to be combined in order to achieve the overall attack, and to this purpose propositional conjunction and disjunction are usually adopted as combinators. On top of this basic model, a number of extensions and applications of attack trees have been proposed, demonstrating how flexible and effective a tool they are in practice. Figure 1 displays a simplistic attack tree, where the overall goal of entering a bank vault is obtained by either bribing a guard or by stealing the combination and neutralising the alarm.

Historically, attack trees are produced manually by teams of experts, known as Red Teams. When considering complex and sizeable systems, however, the manual construction of attack trees becomes error-prone and necessarily not exhaustive. Automated techniques are therefore needed to infer complete and succinct attack trees from formal specifications. Existing approaches, surveyed in Sect. II, all suffer from focusing on computer networks, and thus suggest specification languages tailored to this domain. Moreover, the model-checking techniques that have been proposed recently for generating attack trees lead to an exponential explosion of the state space, limiting the applicability of automated search procedures.

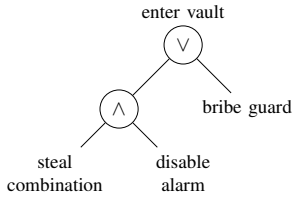



Figure 1. How to enter a bank vault, for dummies.

In order to overcome these drawbacks, we develop a static analysis approach where attack trees are automatically inferred from process algebraic specifications in a syntax-directed fashion. The advantage of resorting to process calculi is manyfold. First, a process algebraic specification requires focusing on the structural and functional definition of a system, and from this deriving the threat scenario automatically, rather than thinking of it from the start, as it seems necessary with existing approaches. Second, the affinity of process calculi with programming languages established their usefulness in the formal design of complex systems, that are described in terms of interacting components. In turn, formal specifications enable the automated verification of behavioural properties at design time, hence before the actual system is produced, matching an ever-growing need for deploying software that lives up to given requirements in terms of security, safety, and performance. Finally, and most importantly, process calculi have proven useful formal languages for describing software systems, organisations, and physical infrastructure in a uniform manner.

The compromise for obtaining such broad a domain coverage while retaining a reasonable expressive power takes place at the level of attack definition. At a high level of abstraction, an attack can be defined as a sequence of actions undertaken by an adversary in order to make unauthorised use of a target asset. In a process algebraic world, this necessary *interaction* between the adversary and the target system is construed in terms of communicating processes. Input actions on the system side can be thought of as *security checks*, that require some information to be fulfilled. In particular, the capability of communicating over a given channel requires the knowledge of the channel itself and of the communication standard. This could include, for example, the knowledge of some cryptographic keys used to secure the communication. Hence, we can think of a compound attack as a set of channels needed to activate the


DOI 10.1109/CSF.2014.31

desired behaviour on the target system side, that is, to reach a particular location that should be secured from unauthorised access. Channels will thus be the basic constituent of attacks in our framework, for the adversary needs such knowledge to interact with the system.
This coarse yet powerful abstraction allows modelling a great many domains. In IT systems, a channel can be thought of as a wired or wireless communication link. In the physical world, a channel can represent a door, and its knowledge the ownership of a suitable key or the capability of bypassing a retinal scan. It is then straightforward to devise *attack metrics* by assigning costs to channels, and then look for attacks that are minimal with respect to the given notion of cost.
On the complexity side, being syntax-driven, static analysis often enjoys better scalability than model-checking approaches. Even though the theoretical complexity of our analysis is still exponential in the worst case, such a price depends on the shape of the process under study, and is not incurred systematically as when a model checker generates the state space.

*Organisation of the paper.* In Sect. III we introduce the calculus that we use as formal specification language. Then, we define a static analysis for automatically inferring attack trees from such specifications.
First, we translate the process $P$ of interest into a set of propositional formulae (Sect. IV), expressing the dependency between the knowledge of channels and the reachability of program points in $P$.
Second, given a location $l$ indicating a program point of interest, we apply a backward-chaining search to these formulae so as to generate an attack tree (Sect. V). Considering all the paths leading to $l$ in $P$, the tree shows what information is required to attain $l$ and how it has to be combined. A tree for label $l$ is condensed into a propositional formula $[\![l]\!]$, whose satisfiability establishes the reachability of $l$ in $P$.
Finally, if a map from atomic attacks (i.e., channels) to costs is provided, we compute the sets of channels that allow reaching $l$ incurring minimal cost, where minimality is sought with respect to an objective function defined on a liberal cost structure. This step reduces to computing the cheapest satisfying assignments of formula $[\![l]\!]$, exploiting a recent application of Satisfiability Modulo Theories (SMT) to optimisation problems (Sect. VI).
A Java implementation of the framework, briefly described in Sect. VII, is available, which takes as input a process in our calculus and a location of interest, and displays graphically the related attack tree. Levering the cost map, the tool also computes the cheapest sets of atomic attacks leading to $l$. The usefulness of the framework is demonstrated on the study of the *NemID* system, a national-scale authentication system used in Denmark to provide secure Internet communication between citizens and public institutions as well as private companies.
Section VIII concludes and sketches a line for future work.

## II. Related Work

Graphical representations of security threats are often used to convey complex information in an intuitive way. Formalisation of such graphical objects are referred to chiefly as *attack graphs* [1], [2], [3], [4] and *attack trees* [5], [6], [7], [8], [9]. In this work we prefer the phrase "attack trees", but our procedure can be adapted to generate attack graphs.
While different authors have different views on the information that should decorate such objects, instrumental to the analysis that the tree or the graph is supporting, all definitions share the ultimate objective of showing how atomic attacks (i.e., the leaves) can be combined to attain a target goal (i.e., the root). This perspective is enhanced in the seminal work of Schneier [5], that found a great many extensions and applications. In particular, Mauw and Oostdijk [7] lay down formal foundations for attack trees, while Kordy et al. [10] and Roy et al. [11] suggest ways to unify attacks and countermeasures in a single view. Even though Schneier's work is mostly credited for having introduced attack trees, and it had certainly a crucial role in making attack trees mainstream in computer security, the origin of this formalism can be traced back to fault trees, expert systems (e.g., Kuang [12]), and privilege graphs [13].
As for the automated generation of attack graphs, the literature is skewed towards the investigation of network-related vulnerabilities: available tools expect as input rich models, including information such as the topology of the network and the set of atomic attacks to be considered. The backward search techniques of Phillips and Swiler [1] and Sheyner et al. [6] have proven useful to cope with the explosion of the state space due to such expressive models. However, the search has to be carried out on a state space that is exponential in the number of system variables, whose construction is the real bottle-neck of these approaches, and the result graph tends to be large even if compact BDD-based representations are used, as argued in [14]. In particular, in [6] a model checking-based approach is developed, where attack graphs are characterised as counter-examples to safety properties; a detailed example is discussed in [3]. Similarly to Phillips and Swiler, we adopt an attacker-centric perspective, which cannot simulate benign system events such as the failure of a component, as in [6]. Directly addressing the exponential blow-up of [6], Ammann et al. [14] propose a polynomial algorithm, but the drop in complexity relies on the assumption of monotonicity of the attacker actions and on the absence of negation. On the same line, Ou et al. [15] present an algorithm which is quadratic in the number of machines in the network under study.
As for the analyses developed on top of attack trees, we present a reachability analysis which computes the cheapest sets of atomic attacks that allow attaining a location of interest in the system, as it is standard in the attack tree literature. This approach seamlessly encompasses the probabilistic analysis of [6], [3] (costs to atomic attacks would represent their likelihood and the objective function would compute the overall probability) and offers a uniform framework to address other quantitative questions [16], [17].
Finally, Mehta et al. [4] present a technique for ranking subgraphs so as to draw attention to the most promising security flaws. Whilst we do not directly tackle this issue, for condensing an entire tree into a formula we gain in performance but we lose the original structure, a post-processing step could be undertaken to compute the value of the internal nodes (subformulae).
The idea of translating process algebraic specifications into sets of logical formulae has been developed in various contexts. In particular, we have been inspired by Blanchet's translation of the applied $\pi$-calculus into first-order formulae [18]. However, the main problem we discuss is propositional; ideas for a first-order extension are sketched in Appendix B.
Our SMT-based solution technique is inspired by [19], which

we extend to symbolic and non-linearly ordered cost structures in [20]. The technique proves useful to answer quantitative questions on attack trees characterised as logical formulae, and therefore we briefly introduce it, the main contribution of this work and its prime focus remaining however the automated generation of attack trees.

It is worthwhile observing that our assumption about channels that provide given degrees of security, embedding for instance cryptographic operations, is supported by a significant line of research on so-called *secure channels* (e.g., see [21]). In particular, when focusing on systems as opposed to protocols, it is reasonable to assume standard mechanisms to be in place for providing various security features. Such mechanisms are mimicked in our work by secure communication channels, and in this sense we say that an attack can be interpreted as a set of such channels.

## III. The Value-Passing Quality Calculus

As specification formalism we adopt the Quality Calculus [22], a process calculus in the $\pi$-calculus family. In particular, the Quality Calculus introduces a new kind of input binders, termed *quality binders*, where a number of inputs are *simultaneously* active, and we can proceed as soon as some of them have been received, as dictated by a guard instrumenting the binder. While these binders enhance the succinctness of highly-branching process-algebraic models, thus increasing their readability, they do not increase the expressiveness of the language, and thus the analysis of Sect. IV carries seamlessly to a variety of process calculi without such binders. Therefore, as far as this work is concerned, the reader can think of the calculus as a broadcast $\pi$-calculus enriched with quality binders.

In the following we present the syntax and discuss informally the semantics of a fragment of the calculus. A formal account of the semantics is deferred to Appendix A, for it is instrumental to show the correctness of the entire approach but does not constitute a novelty per se.

### A. Syntax and intended semantics

The Value-Passing Quality Calculus is displayed in Table I. The calculus consists of four syntactic categories: processes $P$, input binders $b$, terms $t$, and expressions $e$. A process can be prefixed by a restriction, an input binder, an output $c!t$ of term $t$ on channel $c$, or be the terminated process $\mathsf{0}$, the parallel composition of two processes, a replicated process, or a case clause. We write $P \Longrightarrow P'$ to denote that process $P$ evolves to process $P'$, and $\Longrightarrow^*$ for its transitive closure.

A process $c!t.P$ broadcasts term $t$ over channel $c$ and evolves to $P$: all processes ready to make an input on $c$ will synchronise and receive the message, but we proceed to $P$ even if there exists no such process (i.e., broadcast is non-blocking).

An input binder can either be a simple input $c?x$ on channel $c$, or a *quality binder* $\&_q(b_1, \ldots, b_n)$, where the $n$ sub-binders are *simultaneously* active. A quality binder is consumed as soon as enough sub-binders have been satisfied, as dictated by the quality guard $q$. A quality guard can be any boolean predicate: we use the abbreviations $\forall$ and $\exists$ for the predicates specifying that all the sub-binders or at least one sub-binder have to be satisfied before proceeding, respectively. For instance, the process $\&_\exists(c_1?x_1, c_2?x_2).P$ evolves to $P$ as soon as an input is received on $c_1$, or an input is received on $c_2$, or both the sub-binders are satisfied.

Table I. The syntax of the calculus.

| | | | |
|---|---|---|---|
| $P$ | ::= | $\mathsf{0} \mid (\nu c)\,P \mid P_1 \mid P_2 \mid {}^l b.P \mid {}^l c!t.P$ | |
| | $\mid$ | $!P \mid {}^l\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ | (Process) |
| $b$ | ::= | $c?x \mid \&_q(b_1, \cdots, b_n)$ | (Binder) |
| $t$ | ::= | $c \mid y$ | (Term) |
| $e$ | ::= | $x \mid \mathsf{some}(t) \mid \mathsf{none}$ | (Expression) |

In terms of security checks, the existential quality guard describes scenarios in which different ways of fulfilling a check are available, e.g. different ways of proving one's identity. In contrast to this, the universal quality guard $\forall$ describes checks that require a number of sub-conditions to be met at the same time.

When a quality binder is consumed, some input variables occurring in its sub-binders might have not been bound to any value, if this is allowed by the quality guard. This is the case of the previous example:

$$(\nu c_1)(\nu c_2)(\nu c_3)(\&_\exists(c_1?x_1, c_2?x_2).P | c_1!c_3.P')$$

evolves to $P|P'$, even if no input is received on $c_2$. In order to record which inputs have been received and which have not, we always bind an input variable $x$ to an expression $e$, which is $\mathsf{some}(c)$ if $c$ is the name received by the input binding $x$, or $\mathsf{none}$ if the input is not received but we are continuing anyway. In this sense, we say that expressions represent *optional data*, while terms represent *data*, in the wake of programming languages like Standard ML. In order to insist on this distinction, variables $x$ are used to mark places where expressions are expected, whereas variables $y$ stand for terms. Hence, in the previous example we obtain:

$$(\nu c_1)(\nu c_2)(\nu c_3)(\&_\exists(c_1?x_1, c_2?x_2).P | c_1!c_3.P') \Longrightarrow^* (\nu c_1)(\nu c_2)(\nu c_3)(P[\mathsf{some}(c_3)/x_1, \mathsf{none}/x_2] | P')$$

that is, in $P$ variable $x_1$ is replaced by $\mathsf{some}(c_3)$ and $x_2$ is replaced by $\mathsf{none}$.

The case clause in the syntax of processes is then used to inspect whether or not an input variable, bound to an expression, is indeed carrying data. The process $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ evolves into $P_1[c/y]$ if $x$ is bound to $\mathsf{some}(c)$; otherwise, if $x$ is bound to $\mathsf{none}$, $P_2$ is executed. Therefore, case clauses allow detecting which condition triggered passing a binder, that is, how a security check has been fulfilled.

The syntax presented above defines a fragment of the full calculus presented in [22]. First, we have described a value-passing calculus, as channels are syntactically restricted to names; second, terms are pure, that is, only names and variables are allowed (no function application); last, the calculus is limited to test whether or not a given input variable carries data, but cannot test which data it is possibly carrying. We deem that these simplifications match with the abstraction level of the analysis, where secure channels are assumed that have to be established by executing given security protocols, and thus enjoy known properties. For the very same reason we opted for pure terms, since equational reasoning is usually exploited in process calculi to model cryptographic primitives (a Quality Calculus with such a feature is presented in [23]). An extension

of the framework that encompasses the key features of the full calculus is briefly commented upon in Appendix B.
Finally, in order to identify locations of interest in a system, we introduce a non-functional extension to the calculus, where a program point is instrumented with a *unique* label $l \in \mathbb{N}$. This is the case of binders, outputs, and case clauses in Table I: we limit labels to be placed before these constructs for they represent the real actions a process can perform. We say that a label $l$ occurring in a process $P$ is reached in an actual execution when the sub-process $P'$ following $l$ is ready to execute, denoted $C[^l P']$.
As usual, in the following we consider closed processes (no free variable), and we make the assumption that variables and names are bound exactly once, so as to simplify the technicalities without impairing the expressiveness of the framework.

### B. Attacker model

We assume that a system $P$ is deployed in a hostile environment, simulated by an adversary process $Q$ running in parallel with $P$. The ultimate aim of our analysis is to compute an attack tree that shows what channels $Q$ has to communicate over (and thus, to know) in order to drive $P$ to a given location $l$, i.e., $P|Q \Longrightarrow^* C[^l P']$, where $C[^l P']$ denotes a sub-process of $P|Q$ that has reached label $l$.
For an attack tree shows the channels that $Q$ needs in order to attain a program point $l$, the implicit assumption is that the attacker is capable of obtaining any required channel, namely, the attacker is able to fulfil any security check implemented by $P$ on the way to $l$. Obviously, different checks pose different challenges to the attacker, and therefore we temper this essentially qualitative view with a quantification of the *cost* $Q$ incurs to learn a channel, i.e., the effort related to obtaining some information, such as a cryptographic key, or a tool. Therefore, after having computed an attack tree $T_l$ for $l$, our question will be what are the attacks of minimal cost among those described by $T_l$.

### C. Example

Let us introduce now an example that will be developed throughout the paper. *NemID* (literally: EasyID)[1] is an asymmetric cryptography-based log-in solution for on-line banking and public on-line services in Denmark, used by virtually every person who resides in the country. The log-in application is based on a Java applet which is distributed to authorised service providers, and through which their customers can be authenticated. For technological and historical reasons, the applet allows proving one's identity with various sets of credentials. In particular, private citizens can log-in with their social security number, password, and a one-time password, or by exhibiting an X.509-based certificate. Moreover, on mobile platforms that do not support Java, a user is authenticated through a classic id-password scheme.
The system is modelled in the Value-Passing Quality Calculus as follows:

$$NemID \triangleq (\nu \mathsf{login}) \ldots (\nu \mathsf{access})(!Login \mid !Applet \mid !Mobile)$$

$$\begin{aligned}
Applet \triangleq\ & {}^{1}\&_{\exists}(\mathsf{cert}?x_{\mathsf{cert}}, \&_{\forall}(\mathsf{id}?x_{\mathsf{id}}, \mathsf{pwd}?x_{\mathsf{pwd}}, \mathsf{otp}?x_{\mathsf{otp}})).\\
& {}^{2}\mathsf{case}\ x_{\mathsf{cert}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{cert}}) : {}^{3}\mathsf{login!ok}\ \mathsf{else}\\
& \quad {}^{4}\mathsf{case}\ x_{\mathsf{id}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{id}}) :\\
& \quad\quad {}^{5}\mathsf{case}\ x_{\mathsf{pwd}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{pwd}}) :\\
& \quad\quad\quad {}^{6}\mathsf{case}\ x_{\mathsf{otp}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{otp}}) : {}^{7}\mathsf{login!ok}\ \mathsf{else}\ 0\\
& \quad\quad \mathsf{else}\ 0\\
& \quad \mathsf{else}\ 0
\end{aligned}$$

$$\begin{aligned}
Mobile \triangleq\ & {}^{8}\&_{\forall}(\mathsf{id}?x'_{\mathsf{id}}, \mathsf{pin}?x_{\mathsf{pin}}).\\
& {}^{9}\mathsf{case}\ x'_{\mathsf{id}}\ \mathsf{of}\ \mathsf{some}(y'_{\mathsf{id}}) :\\
& \quad {}^{10}\mathsf{case}\ x_{\mathsf{pin}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{pin}}) : {}^{11}\mathsf{login!ok}\ \mathsf{else}\ 0\\
& \quad \mathsf{else}\ 0
\end{aligned}$$

$$Login \triangleq\ {}^{12}\mathsf{login}?x.{}^{13}\mathsf{access!ok}$$

The system consists of three processes running in parallel an unbounded number of times. For the sake of brevity, we have omitted to list all the restrictions in front of the parallel components, that involve all the names occurring in the three processes.
Process *Login* is in charge of granting access to the system: whenever a user is authenticated via the applet or a mobile app, an output on channel login is triggered, which is received at label 12 leading to the output at label 13, which simulates a successful authentication.
Process *Applet* models the applet-based login solution, where login is granted (simulated by the outputs at label 3 and 7) whenever the user exhibits a valid certificate or the required triple of credentials. The quality binder at label 1 implements such a security check: in order to pass the binder, either ($\exists$) a certificate has to be provided, simulated by the first sub-binder, or three inputs have to be received ($\forall$), mimicking an id (id), a password (pwd), and a one-time password (otp). This behaviour is obtained by nesting a universal quality binder in a binder instrumented with an existential quality guard. Observe how case constructs are used to determine what combination allowed passing the binder: at label 2 we check whether the certificate is received, and if this is not the case then we check that the other condition is fulfilled. The main abstraction of our approach takes place at this level, as we can only test whether something is received on a given channel, but we cannot inspect the content of what is received. In other words, the knowledge of channel cert mimics the capability of producing a valid certificate, and thus we say that the semantic load of the communication protocol is shifted onto the notion of secure channel. Observe that this perspective seamlessly allows reasoning about the cost of attacking the system: to communicate over cert, an adversary has to get hold of a valid certificate, e.g. bribing someone or brute-forcing a cryptographic scheme, and this might prove more expensive than guessing the triple of credentials necessary to achieve authentication along the alternative path.
Finally, process *Mobile* describes the intended behaviour of the mobile login solution developed by some authorities (e.g. banks, public electronic mail system), where an id and a password or pin have to be provided upon login.

In the remainder of the paper, we show how an attack tree is inferred automatically given a process $P$ and a label $l$, according to the following plan:

1) $P$ is translated into a set containing propositional formulae, $(i)$ stating the dependency between the

[1] https://www.nemid.nu/dk-en/

knowledge of channels, and $(ii)$ expressing the relationship between such knowledge and the reachability of locations (Sect. IV);

2) backward chaining the formulae representing $P$, a formula $[\![l]\!]$ is synthesised, stating what channels have to be in the knowledge of $Q$ so as to drive $P$ to $l$; a parse tree of such formula is an attack tree showing the combinations of channels that allow reaching $l$ (Sect. V);
3) given a map from channels to costs, we compute the set of minimal-cost attacks that allow reaching $l$ among those described by the tree. This is achieved by solving a sequence of SMT problems (Sect. VI).

Each step will be demonstrated on the *NemID* example introduced above.

## IV. From Processes to Propositional Formulae

The recursive function $[\![P]\!]\mathsf{tt}$, defined in Table II, translates a process $P$ into a set containing propositional formulae of the form $\varphi \Rightarrow \overline{p}$, where $\varphi$ is a propositional formula and $\overline{p}$ is an atom. In the following, we refer to $\varphi$ as the antecedent and to $\overline{p}$ as the consequent of the implication. The semantics of one such formula prescribes that if $\varphi$ evaluates to true ($\mathsf{tt}$) under the knowledge of the adversary $Q$, then $Q$ can obtain $p$ with no additional effort. In other words, if $Q$ can fulfil the security checks described by $\varphi$, then it can also satisfy the check represented by $p$. Technically, this approach exploits an inductive definition of the attacker knowledge, first introduced in connection to security protocols verification.
In $\varphi \Rightarrow \overline{p}$, the consequent $p$ is either

- a channel $c$, atom $\overline{c}$ expressing whether or not $Q$ can get hold of $c$: if $\overline{c} = \mathsf{tt}$ then $c$ is known to the attacker, otherwise if $\overline{c} = \mathsf{ff}$ it is not;
- an input variable $x$, $\overline{x}$ expressing whether $x$ is bound to some value ($\overline{x} = \mathsf{tt}$) or to $\mathsf{none}$ ($\overline{x} = \mathsf{ff}$), that is, atom $\overline{x}$ accounts for the capability of $Q$ of satisfying the input binding $x$;
- a label $l$, atom $\overline{l}$ expressing whether or not $Q$ can make $P$ reach $l$.

In each step of the evaluation, the first parameter of $[\![\cdot]\!]\cdot$ corresponds to the sub-process of $P$ that has still to be translated, while the second parameter is a logic formula $\Phi$, intuitively carrying the hypothesis on the knowledge $Q$ needs to attain the current point in $P$. The translation function is structurally defined over processes. The initial invocation $[\![P]\!]\mathsf{tt}$ assumes that $P$ is executable, thus setting $\Phi = \mathsf{tt}$.
If $P$ is the terminated process $\mathsf{0}$, then there is no location to be attained and thus no formula is produced. If $P$ is a replicated process $!P'$, then the attacker does not need any knowledge in order to make $P$ reach $P'$, as it evolves spontaneously whenever necessary. Similarly, if $P$ is the process $(\nu c)\,P'$, then the attacker does not need any knowledge in order to make $P$ reach $P'$. Parallel processes are translated taking the union of the sets into which the components are translated.
Communication actions and $\mathsf{case}$ clauses determine instead what $Q$ has to know, for we construe input actions as security checks. Whenever the translation reaches a labelled action $a$, that is, $[\![^l a.P']\!]\Phi$, then a formula $\Phi \Rightarrow \overline{l}$ is generated, denoting that $l$ is reached if the adversary's knowledge satisfies $\Phi$ (the attacker can fulfil all the security checks on at least one path to $l$). Moreover, in order to pass $a$ and attain $P'$, $Q$ has to comply with some further requirements, depending on the nature of $a$, and some additional formulae may be produced.
The translation of a binder $^l b.P'$ has two effects. First, some conditions have to be met for reaching $P'$, i.e., the security checks expressed by $b$ have to be fulfilled: this is taken care of by function $\mathsf{hp}(b)$, that given a binder derives the combinations of channels that allows passing it, expressed as a propositional formula; such formula will then extend the hypothesis $\Phi$. Second, whenever $b$ is passed, the adversary gains some knowledge about the content of the input variables defined by the binder: if a binder is satisfied, then some of the input variables receive values other than $\mathsf{none}$; this is taken care of by function $\mathsf{th}(\Phi, b)$.
Consider a simple input $^l c?x.P'$: in order to make $P$ consume the input and evolve to $P'$, the adversary has to know channel $c$, and thus $\mathsf{hp}(c?x) = \overline{c}$ is added to the hypothesis $\Phi$ in $[\![P']\!](\Phi \wedge \overline{c})$. Moreover, once a simple input is consumed, it must be the case that the input variable $x$ has been bound to $\mathsf{some}(c')$, hence we generate the formula $(\Phi \wedge \overline{c}) \Rightarrow \overline{x}$: if $Q$ can reach the input $^l c?x.P'$ and $Q$ knows $c$, then the input can be satisfied. This idea is generalised for quality binders, where the quality guard $q$ determines how to compose the constraints related to the sub-binders. The last section of Table II shows some cases for $q$, but in general any boolean predicate can be used as guard.
When an output $^l c!t$ is encountered, the knowledge of $Q$ increases, as the attacker can now control $c$ at the price of satisfying the hypothesis for reaching the output, and thus a formula $\Phi \Rightarrow \overline{c}$ is generated. In other words, satisfying the checks prescribed by $\Phi$, the attacker automatically fulfils $c$. It is worthwhile observing that this choice is justified by the broadcast semantics, and by the fact that the calculus is limited to testing whether or not something has been received over a given channel, shifting the semantic load on the notion of secure channel.
Finally, a $\mathsf{case}$ construct is translated by taking the union of the formulae into which the two branches are translated: for the check is governed by the content of the $\mathsf{case}$ variable $x$, we record that the $\mathsf{then}$ branch is followed only when $x$ is bound to $\mathsf{some}(c)$ by adding a literal $\overline{x}$ to the hypothesis, as we do for inputs, and we add $\neg\overline{x}$ if the $\mathsf{else}$ branch is followed. Observe that the mutual exclusion of the two branches is accounted for by the conjunction of the hypothesis with $\overline{x}$ and $\neg\overline{x}$, respectively. This approach is equivalent to consider $\mathsf{xor}$ nodes in the tree, as in [24].

In Appendix C-A we show that the cardinality of the set of formulae $[\![P]\!]\mathsf{tt}$ is linear in the number of actions $n$ occurring in $P$, while the number of literals occurring in the formulae grows with $n^2$, and we argue that this a precise theoretical bound. Nonetheless, the examples arising from the *NemID* system and other realistic scenarios suggest a better behaviour than a quadratic growth.
Moreover, in the appendix we also show the following result, that has a crucial role in the correctness of the backward-chaining procedure of Sect. V.

**Lemma 1.** *Let $P$ be a closed process in the Value-Passing Quality Calculus, and assume that each variable $x$ and name $c$ occurring in $P$ is bound exactly once. Then, for any variable $x$*

Table II. THE TRANSLATION FROM PROCESSES TO PROPOSITIONAL FORMULAE.

$$
\begin{array}{lcl}
[\![0]\!]\Phi & = & \emptyset \\
[\![!P]\!]\Phi & = & [\![P]\!]\Phi \\
[\![(\nu c)\,P]\!]\Phi & = & [\![P]\!]\Phi \\
[\![P_1|P_2]\!]\Phi & = & [\![P_1]\!]\Phi \cup [\![P_2]\!]\Phi \\
[\![^l b.P]\!]\Phi & = & [\![P]\!](\Phi \wedge \mathsf{hp}(b)) \cup \mathsf{th}(\Phi, b) \cup \{\Phi \Rightarrow \overline{l}\} \\
[\![^l c!t.P]\!]\Phi & = & [\![P]\!]\Phi \cup \{\Phi \Rightarrow \overline{c}\} \cup \{\Phi \Rightarrow \overline{l}\}
\end{array}
$$

$$
[\![^l \mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y)\colon P_1\ \mathsf{else}\ P_2]\!]\Phi = [\![P_1]\!](\Phi \wedge \overline{x}) \cup [\![P_2]\!](\Phi \wedge \neg\overline{x}) \cup \{\Phi \Rightarrow \overline{l}\}
$$

$$
\begin{array}{l}
\mathsf{hp}(c?x) = \overline{c} \\
\mathsf{hp}(\&_q(b_1, \ldots, b_n)) = [\{q\}](\mathsf{hp}(b_1), \ldots, \mathsf{hp}(b_n)) \\
\mathsf{th}(\Phi, c?x) = \{(\Phi \wedge \overline{c}) \Rightarrow \overline{x}\} \\
\mathsf{th}(\Phi, \&_q(c_1?x_1, \ldots, c_n?x_n)) = \bigcup_{i=1}^{n} \mathsf{th}(\Phi, c_i?x_i)
\end{array}
$$

$$
\begin{array}{l}
[\{\forall\}](\varphi_1, \ldots, \varphi_n) = \bigwedge_{i=1}^{n} \varphi_i \\
[\{\exists\}](\varphi_1, \ldots, \varphi_n) = \bigvee_{i=1}^{n} \varphi_i
\end{array}
$$

*in $P$, there exists exactly one formula $\varphi \Rightarrow \overline{x}$ in the translation $[\![P]\!]\mathsf{tt}$, and $\overline{x}$ does not occur in $\varphi$.*

Finally, a similar result holds for every label $l$ in $P$, as labels are unique and the translation encounters $l$ exactly once.

### A. Example

Consider the process *NemID* discussed in Sect. III-C, and its translation $[\![NemID]\!]\mathsf{tt}$. Applying the rules of Table II, we obtain the union of the following sets of formulae ($\mathsf{tt}$ conjuncts are omitted).

$$
\begin{array}{l}
[\![Applet]\!]\mathsf{tt} = \{ \\
\quad \overline{1}, \\
\quad \overline{\mathsf{cert}} \Rightarrow \overline{x}_{\mathsf{cert}}, \\
\quad \overline{\mathsf{id}} \Rightarrow \overline{x}_{\mathsf{id}}, \\
\quad \overline{\mathsf{pwd}} \Rightarrow \overline{x}_{\mathsf{pwd}}, \\
\quad \overline{\mathsf{otp}} \Rightarrow \overline{x}_{\mathsf{otp}}, \\
\quad \underbrace{\overline{\mathsf{cert}} \vee (\overline{\mathsf{id}} \wedge \overline{\mathsf{pwd}} \wedge \overline{\mathsf{otp}})}_{\varphi} \Rightarrow \overline{2}, \\
\quad \varphi \wedge \overline{x}_{\mathsf{cert}} \Rightarrow \overline{3}, \\
\quad \varphi \wedge \overline{x}_{\mathsf{cert}} \Rightarrow \overline{\mathsf{login}}, \\
\quad \varphi \wedge (\neg\overline{x}_{\mathsf{cert}}) \Rightarrow \overline{4}, \\
\quad \varphi \wedge (\neg\overline{x}_{\mathsf{cert}}) \wedge \overline{x}_{\mathsf{id}} \Rightarrow \overline{5}, \\
\quad \varphi \wedge (\neg\overline{x}_{\mathsf{cert}}) \wedge \overline{x}_{\mathsf{id}} \wedge \overline{x}_{\mathsf{pwd}} \wedge \overline{x}_{\mathsf{otp}} \Rightarrow \overline{7}, \\
\quad \varphi \wedge (\neg\overline{x}_{\mathsf{cert}}) \wedge \overline{x}_{\mathsf{id}} \wedge \overline{x}_{\mathsf{pwd}} \wedge \overline{x}_{\mathsf{otp}} \Rightarrow \overline{\mathsf{login}} \\
\}
\end{array}
$$

$$
\begin{array}{l}
[\![Mobile]\!]\mathsf{tt} = \{ \\
\quad \overline{8}, \\
\quad \overline{\mathsf{id}} \Rightarrow \overline{x'}_{\mathsf{id}}, \\
\quad \overline{\mathsf{pin}} \Rightarrow \overline{x}_{\mathsf{pin}}, \\
\quad \overline{\mathsf{id}} \wedge \overline{\mathsf{pin}} \Rightarrow \overline{9}, \\
\quad \overline{\mathsf{id}} \wedge \overline{\mathsf{pin}} \wedge \overline{x'}_{\mathsf{id}} \Rightarrow \overline{10}, \\
\quad \overline{\mathsf{id}} \wedge \overline{\mathsf{pin}} \wedge \overline{x'}_{\mathsf{id}} \wedge \overline{x}_{\mathsf{pin}} \Rightarrow \overline{11}, \\
\quad \overline{\mathsf{id}} \wedge \overline{\mathsf{pin}} \wedge \overline{x'}_{\mathsf{id}} \wedge \overline{x}_{\mathsf{pin}} \Rightarrow \overline{\mathsf{login}} \\
\}
\end{array}
$$

$$
\begin{array}{l}
[\![Login]\!]\mathsf{tt} = \{ \\
\quad \overline{12}, \\
\quad \overline{\mathsf{login}} \Rightarrow \overline{x}, \\
\quad \overline{\mathsf{login}} \Rightarrow \overline{13}, \\
\quad \overline{\mathsf{login}} \Rightarrow \overline{\mathsf{access}} \\
\}
\end{array}
$$

Notice how each formula models the checks on a given path: for being granted access, i.e., reaching label 13, a communicating process has to know channel login. In turn, other formulae describe what is needed in order to get hold of such information, giving rise to a backward search formalised in Sect. V. Therefore, the translation accounts for the ways a given location or piece of knowledge can be obtained *playing according to the system rules*. To make an analogy, this approach resembles the perfect cryptography assumption often made in security protocol verification, stating that an encrypted term can be decrypted only knowing the corresponding cryptographic key. Similarly, in the example we are saying that the only way for logging in is to possess a set of required credentials: the only way for getting login is to satisfy the checks on one path the leads to issuing login.

### B. Compositionality

It is worthwhile highlighting the modularity of process algebraic specification, which results in a high degree of flexibility when analysing complex systems. In the example above, for instance, while the Java applet is developed by a national contractor, and hence is common to all service providers, each company offers its own mobile app. Assume that a new way to access the system were offered by a bank, which would authenticate a user via a phone number:

$$
\begin{array}{l}
\mathit{Phone} \triangleq \\
\quad {}^{14}\mathsf{phone}?x_{\mathsf{ph}}.{}^{15}\mathsf{case}\ x_{\mathsf{ph}}\ \mathsf{of}\ \mathsf{some}(y_{\mathsf{ph}})\colon {}^{16}\mathsf{login}!\mathsf{ok}\ \mathsf{else}\ 0
\end{array}
$$

$$
\begin{array}{l}
\mathit{NemID}' \triangleq (\nu\mathsf{login}) \ldots (\nu\mathsf{phone}) \\
\qquad\qquad (!\mathit{Login} \mid !\mathit{Applet} \mid !\mathit{Mobile} \mid !\mathit{Phone})
\end{array}
$$

Then we have $[\![\mathit{NemID}']\!]\mathsf{tt} = [\![\mathit{NemID}]\!]\mathsf{tt} \cup [\![\mathit{Phone}]\!]\mathsf{tt}$, that is, the translation of a new top-parallel process is independent from the formulae that have already been generated. Obviously, due care has to be paid to names, e.g., name login in *Phone* has to be the same used in *NemID*. However, while restrictions play a crucial role in the semantics, they are simply ignored by the translation.

Besides being flexible with respect to the analysis of new components, the translation suitably integrates in a refinement cycle, where we start from a coarse abstraction of the system and then progressively refine those components that are revealed as candidates for being attacked, by replacing the corresponding set of formulae with a finer one. The constraint

on names translates to a constraint on the interface of the component: if process $A$ is replaced by process $B$, then $B$ must be activated by the same inputs that activate $A$, and vice-versa it must produce the same outputs towards the external environment that $A$ is producing.

## V. From Formulae to Attack Trees

Once a process has been translated into propositional formulae, it is possible to build an attack tree for each program point automatically, showing what information has to be obtained and how it has to be combined in order to attain the desired goal.

Given a process $P$ and a label $l$ occurring in $P$, we generate a formula $[\![l]\!]$ representing the attack "$l$ is reached" by backward chaining the formulae in $[\![P]\!]\mathsf{tt}$ so as to derive $\bar{l}$.

Before explaining the algorithm, it is worthwhile discussing the nature of the backward chaining-like procedure defined in the following. Standard backward chaining [25, Chp. 7] combines Horn clauses so as to check whether a given goal follows from the knowledge base. Instead, we are in fact trying to derive all the knowledge bases that allow inferring the goal given the inference rules $[\![P]\!]\mathsf{tt}$, which are not strict Horn clauses as they can contain more than one positive literal. Using a slightly different terminology, we are here interested in computing all the *implicants* of $\bar{l}$, while in Sect. VI we will show how to select those implicants that are minimal with respect to a given notion of cost (i.e., *prime* implicants).

The rules for generating $[\![l]\!]$ are displayed in Table III. For our formulae are propositional, there is no unification other than syntactical identity of literals involved in the procedure. Notice that the algorithm only applies valid inference rules.

Rule (Sel) selects the antecedent of the formula leading to the goal $\bar{l}$: since there is a unique such rule, in order to derive $\bar{l}$ we have to derive the antecedent $\varphi$ of $\varphi \Rightarrow \bar{l}$. Observe that we are not interested in deriving $\bar{l}$ in any other way.

Rule (Pone-c) encodes either a tautology (if $\bar{c}$ has to be inferred then $\bar{c}$ is in the knowledge base) or applications of modus ponens ($\bar{c}$ is derived assuming $\varphi$, thanks to $\varphi \Rightarrow \bar{c}$): the whole rule is an instance of disjunction introduction. This is the point where our algorithm differs from plain backward chaining: since we are building the knowledge bases that allow inferring $\bar{l}$, whenever we encounter a literal $\bar{c}$ we need to account for all the ways of deriving $\bar{c}$, namely by placing $\bar{c}$ itself in the knowledge base or by satisfying a rule whose consequent is $\bar{c}$. Similarly, rule (Pone-x) encodes an application of modus ponens, taking advantage of the uniqueness of $\varphi \Rightarrow \overline{x}$ (cf. Lemma 1).

Rules (Tolle-) collect applications of modus tollens, in the classic backward fashion (i.e., when considering the derivation from the leaves to the root such steps would encode that modus). Rules (DM-) encode De Morgan's laws. Finally, rules (Comp-) simply state the compositionality of the procedure.

It is worthwhile observing that in classic backward chaining loops are avoided by checking whether a new sub-goal (i.e., literal to be derived) is already on the goal stack (i.e., is currently being derived): component $\mathcal{D}$ in Table III is in charge of keeping track of the current goals, but this is done on a local basis as opposed to the traditional global stack, that would result if $\mathcal{D}$ were treated as a global variable. As shown in Remark 1 below, in our settings the global stopping criterion would lead to unsound results. Moreover, observe that using the local environment $\mathcal{D}$ we lose the linear complexity in $|[\![P]\!]\mathsf{tt}|$ typical of backward chaining, and incur an exponential complexity in the worst case. Nonetheless, mind to observe that this theoretical bound is not incurred systematically.

Table III. How to synthesise a propositional formula describing the attack tree $T_l$.

$$[\![l]\!] = [\![\varphi]\!]\emptyset \qquad \text{where } (\varphi \Rightarrow \bar{l}) \in [\![P]\!]\mathsf{tt} \qquad \text{(Sel)}$$

$$[\![\overline{c}]\!]\mathcal{D} = \overline{c} \ \vee \begin{cases} \bigvee_{\{\varphi \mid (\varphi \Rightarrow \overline{c}) \in [\![P]\!]\mathsf{tt}\}} [\![\varphi]\!](\mathcal{D} \cup \{\overline{c}\}) & \text{if } \overline{c} \notin \mathcal{D} \\ \mathsf{ff} & \text{otherwise} \end{cases} \quad \text{(Pone-c)}$$

$$[\![\neg\overline{c}]\!]\mathcal{D} = \begin{cases} \bigwedge_{\{\varphi \mid (\varphi \Rightarrow \overline{c}) \in [\![P]\!]\mathsf{tt}\}} [\![\neg\varphi]\!](\mathcal{D} \cup \{\neg\overline{c}\}) & \text{if } \neg\overline{c} \notin \mathcal{D} \\ \mathsf{tt} & \text{otherwise} \end{cases} \quad \text{(Tolle-c)}$$

$$[\![\overline{x}]\!]\mathcal{D} = [\![\varphi]\!]\mathcal{D} \qquad \text{where } (\varphi \Rightarrow \overline{x}) \in [\![P]\!]\mathsf{tt} \qquad \text{(Pone-x)}$$

$$[\![\neg\overline{x}]\!]\mathcal{D} = [\![\neg\varphi]\!]\mathcal{D} \qquad \text{where } (\varphi \Rightarrow \overline{x}) \in [\![P]\!]\mathsf{tt} \qquad \text{(Tolle-x)}$$

$$[\![\neg(\varphi_1 \wedge \cdots \wedge \varphi_n)]\!]\mathcal{D} = [\![\neg\varphi_1]\!]\mathcal{D} \vee \cdots \vee [\![\neg\varphi_n]\!]\mathcal{D} \quad \text{(DM-1)}$$
$$[\![\neg(\varphi_1 \vee \cdots \vee \varphi_n)]\!]\mathcal{D} = [\![\neg\varphi_1]\!]\mathcal{D} \wedge \cdots \wedge [\![\neg\varphi_n]\!]\mathcal{D} \quad \text{(DM-2)}$$

$$[\![\varphi_1 \wedge \cdots \wedge \varphi_n]\!]\mathcal{D} = [\![\varphi_1]\!]\mathcal{D} \wedge \cdots \wedge [\![\varphi_n]\!]\mathcal{D} \quad \text{(Comp-1)}$$
$$[\![\varphi_1 \vee \cdots \vee \varphi_n]\!]\mathcal{D} = [\![\varphi_1]\!]\mathcal{D} \vee \cdots \vee [\![\varphi_n]\!]\mathcal{D} \quad \text{(Comp-2)}$$

$$[\![\mathsf{tt}]\!]\mathcal{D} = \mathsf{tt} \qquad [\![\mathsf{ff}]\!]\mathcal{D} = \mathsf{ff}$$

Notice that, in virtue of Lemma 1, we do not need to keep track of literals $\overline{x}$ in $\mathcal{D}$, as we cannot meet with a cycle.

Finally, observe that a parse tree $T_l$ of $[\![l]\!]$ is an attack tree, showing how $l$ can be attained by combining the knowledge of given channels. The internal nodes of the tree contain a boolean operator in $\{\wedge, \vee\}$, while the leaves contain literals representing the knowledge of channels. As De Morgan's laws are used to push negations to literals of $[\![l]\!]$, negation can only occurs in the leaves of $T_l$. This approach is in line with the literature, where propositional formulae are interpreted as denotations of attack trees (e.g., see [8]). In the following, we shall manipulate attack trees always at their denotation level.

For the sake of discussion, it is worthwhile noticing that the procedure for generating $[\![l]\!]$ can be used to generate a tree explicitly during the computation, or even an And-Or graph [25, Chp. 4]. It is unclear to us, however, whether the more compact graph representation would be simpler to understand.

### A. Formal correctness

Appendix C-B gives a formal account of the soundness of the framework, first showing the correctness of the backward-chaining procedure, and then connecting it to the semantics of the calculus.

The formal correctness of the backward-chaining procedure is discussed in Theorem 1. The theorem argues that a model $m$ satisfying $[\![l]\!]$ (that is, an assignments to atoms of $[\![l]\!]$ such that the formula evaluates to $\mathsf{tt}$) is a knowledge base that allows deriving $\bar{l}$ using the formulae in $[\![P]\!]\mathsf{tt}$ as inference rules (together with classic propositional rules). Hence, whenever we want to establish the viability of the attack tree $T_l$, we can limit to look for models of $[\![l]\!]$. Section VI explains how such models can be computed.

Moreover, to complete the picture, we need to establish the correctness of $[\![l]\!]$ with respect to the reachability of $l$ in the semantics of process $P$. This is formalised in Theorem 2. The central idea is that a model $m$ of $[\![l]\!]$ denotes a set of channels $\{c_1, \ldots, c_n\}$ (such that $m(\overline{c_i}) = \mathsf{tt}$), and these channels are sufficient for satisfying all the inputs performed in $P$ along at least one path to $l$. In other words, if the adversary $Q$ can perform outputs on $\{c_1, \ldots, c_n\}$, then $P|Q \Longrightarrow^* C[^l P']$.
It is worthwhile noticing that the correctness of the analysis with respect to the semantics corresponds to the *exhaustiveness* of attack trees as defined in [**?**]: $[\![l]\!]$ covers all possible attacks leading to $l$.
Finally, observe that $[\![l]\!]$ only contains literals $\overline{c}$ corresponding to channels, that is, the backward chaining-like procedure described above and formalised in Table III eliminates all the literals $\overline{x}$. Therefore, the reachability of $l$ is only expressed in terms of knowledge of channels. This result is formalised in Lemma 3, and will be leveraged in Sect. VI in order to guarantee that a map from channels to cost suffices to quantify an attack.

### B. Example

Consider the process *NemID* discussed above and its translation $[\![\textit{NemID}]\!]\mathsf{tt}$. Label $13$ marks the point where a user is authenticated into the system, and therefore is a location of interest for our analysis. Figure 2(a) shows the attack tree $T_{13}$, as generated by our implementation, presented in Sect. VII. The backward-chaining procedure takes about 1 second on an ordinary laptop. The denotation of $T_{13}$ is given by the following formula:

$$\begin{aligned}[\![13]\!] \;=\; & \overline{\mathsf{login}} \;\vee\; \big((\overline{\mathsf{cert}} \vee (\overline{\mathsf{id}} \wedge \overline{\mathsf{pwd}} \wedge \overline{\mathsf{otp}})) \wedge \overline{\mathsf{cert}}\big) \;\vee \\ & \big((\overline{\mathsf{cert}} \vee (\overline{\mathsf{id}} \wedge \overline{\mathsf{pwd}} \wedge \overline{\mathsf{otp}})) \wedge (\neg\overline{\mathsf{cert}}) \\ & \qquad\qquad \wedge\; \overline{\mathsf{id}} \wedge \overline{\mathsf{pwd}} \wedge \overline{\mathsf{otp}}\big) \;\vee \\ & \big(\overline{\mathsf{id}} \wedge \overline{\mathsf{pin}}\big)\end{aligned}$$

As a matter of fact, the algorithm tends to generate simple but redundant formulae, that can be simplified automatically, e.g. via a reduction to a normal form. The following formula, for instance, is equivalent to $[\![13]\!]$ but highlights more clearly the ways in which an attack can be carried out:

$$\overline{\mathsf{login}} \;\vee\; (\overline{\mathsf{id}} \wedge \overline{\mathsf{pin}}) \;\vee\; (\overline{\mathsf{id}} \wedge \overline{\mathsf{pwd}} \wedge \overline{\mathsf{otp}}) \;\vee\; \overline{\mathsf{cert}}$$

Observe that the formula above is in Disjunctive Normal Form (DNF). Such normal form has the merit of providing an immediate intuition of the alternative conditions that lead to attain the program point under study, as displayed in Fig. 2(b). However, the conversion to DNF may cause an exponential blow-up in the number of literals, and compact translations require to introduce fresh atoms, garbling the relation between the tree and the original system. Therefore, we did not implement such conversion in our tool.
Finally, notice that the disjunct $\overline{\mathsf{login}}$ encodes the possibility of obtaining a login token in any other way not foreseen in the system, and thus accounts for all the attacks not explicitly related to the shape of our formalisation. Such component can be disregarded by assigning it the maximum possible cost, as we shall see in the next section.

**Remark 1.** *We conclude this section showing why the global stopping criterion is unsound for the procedure of Table III. Consider the following set of formulae:*

$$\overline{a} \Rightarrow \overline{b} \qquad \overline{b} \Rightarrow \overline{a} \qquad \overline{a} \wedge \overline{b} \Rightarrow \overline{7}$$

*which stems from a conveniently simplified translation of the process*

$$P \triangleq \; {}^{1}a?x_a.{}^{2}b!b \,|\, {}^{3}b?x_b.{}^{4}a!a \,|\, {}^{5}a?x'_a.{}^{6}b?x'_b.{}^{7}c!c$$

*The generation of $[\![7]\!]$ unfolds as follows:*

$$[\![7]\!] = [\![\overline{a} \wedge \overline{b}]\!]\emptyset = [\![\overline{a}]\!]\emptyset \wedge [\![\overline{b}]\!]\emptyset$$

*where, in particular, it is*

$$\begin{aligned}[\![\overline{a}]\!]\emptyset &= \overline{a} \vee [\![\overline{b}]\!]\{\overline{a}\} = \overline{a} \vee \overline{b} \vee [\![\overline{a}]\!]\{\overline{a}, \overline{b}\} = \overline{a} \vee \overline{b} \vee \mathsf{ff} = \overline{a} \vee \overline{b} \\ [\![\overline{b}]\!]\emptyset &= \overline{b} \vee [\![\overline{a}]\!]\{\overline{b}\} = \overline{b} \vee \overline{a} \vee [\![\overline{b}]\!]\{\overline{b}, \overline{a}\} = \overline{b} \vee \overline{a} \vee \mathsf{ff} = \overline{b} \vee \overline{a}\end{aligned}$$

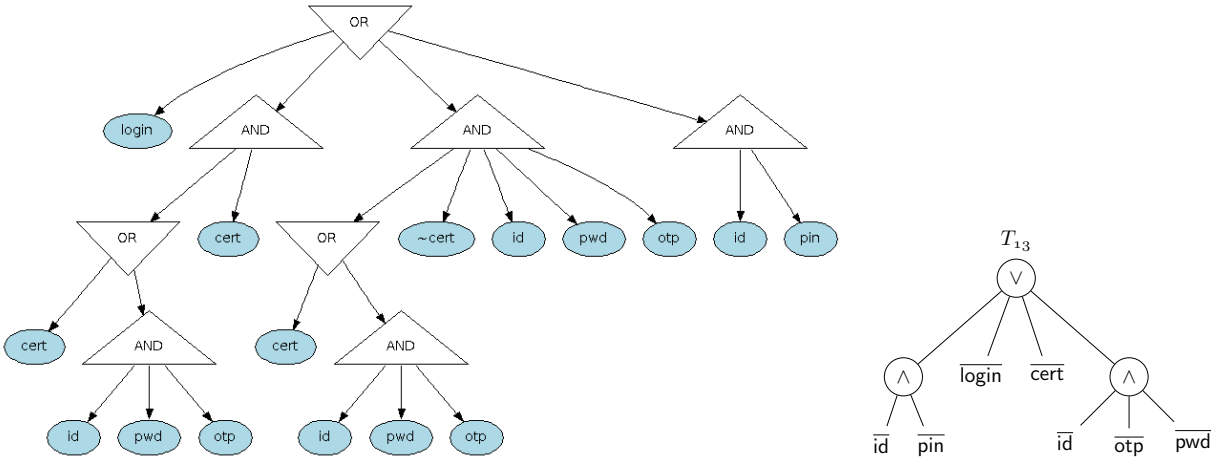
*leading to $[\![7]\!] = \overline{a} \vee \overline{b}$, which is consistent with the reachability of label $7$ in $P$.*
*Assume now to carry out the generation of $[\![7]\!]$ applying a global stopping criterion, that is, to keep track of derived goals in a global environment, initially empty. We would obtain:*

$$[\![\overline{a}]\!]\emptyset = \overline{a} \vee [\![\overline{b}]\!]\{\overline{a}\} = \overline{a} \vee \overline{b} \vee [\![\overline{a}]\!]\{\overline{a}, \overline{b}\} = \overline{a} \vee \overline{b} \vee \mathsf{ff} = \overline{a} \vee \overline{b}$$

*at this point, however, the environment contains $\overline{a}, \overline{b}$, and thus the generation of $[\![\overline{b}]\!]$ leads to $\overline{b}$, resulting in $[\![7]\!] = (\overline{a} \vee \overline{b}) \wedge \overline{b}$, which is not satisfied by the model where only $\overline{a}$ is $\mathsf{tt}$, and thus is wrong. Analogously, we would obtain a wrong result if we chose to unfold $[\![\overline{b}]\!]$ before $[\![\overline{a}]\!]$.*

## VI. Assessing Attack Trees

A number of quantitative problems have been defined on attack trees [17]. Once a tree is characterised as a logical formula, however, a great many of them can be reduced to the problem of computing a satisfying assignment that is minimal (or, dually, maximal) with respect to a given notion of cost. There exist various techniques to cope with such an optimisation problem. In case of numerical costs, for example, the problem can be encoded as a Pseudo-Boolean or an Answer Set Programming optimisation problem. We resort to a more general encoding into Satisfiability Modulo Theories (SMT), as it allows to consider symbolic and non-linearly ordered cost structures.
For $[\![l]\!]$ only contains literals related to the knowledge of channels, we attach the concept of cost to these objects, representing the effort to be paid for obtaining the channel. For instance, an adversary can obtain a secure wireless channel by breaking a given encryption scheme, whose cost can be quantified as number of bits of cryptographic keys, or as time. In the physical world, a channel may represent a door, the cost of passing it being quantifiable in grams of dynamite or symbolically in the lattice $\mathsf{easy} \sqsubset \mathsf{medium} \sqsubset \mathsf{difficult}$, accounting for the strength of the door.
Technically, a map $\mathsf{cost} : \mathit{Names} \to \mathcal{K}$ is introduced, where *Names* is the set of names in the process under study (channels), $(\mathcal{K}, \sqsubseteq)$ is a complete lattice, and $(\mathcal{K}, \oplus)$ is a monoid, $\oplus$ expressing the way costs are combined along a path. We require $\oplus$ to be extensive and monotone [26], and we assume that the least element $\bot$ of $\mathcal{K}$ is the neutral element for $\oplus$.
It is worthwhile noticing how this method seamlessly lends itself to modelling insiders: by setting to $\bot$ the cost of a given channel, we are stating that the adversary is free to use the channel incurring no cost, as if it were already known.




(a) $T_{13}$ as displayed by the Quality Tree Generator, presented in Sect. VII.

(b) The simplified DNF attack tree.

Figure 2. The attack tree $T_{13}$ of the running example.

Similarly, we can model the scenario in which the adversary partially knows a secret, for instance the first character of a password, by decreasing its cost. For the sake of simplicity, we assume that the costs of channels related to literals occurring in $[\![l]\!]$ are independent.
Finally, observe that the quest for assignments of *minimal* cost integrates with the backward-chaining procedure of Sect. V, that avoids derive a literal twice in the same sub-tree. In this sense, our analysis is qualitative with respect to the number of attempts are made to guess a channel: whenever the adversary decides to incur the corresponding cost, a channel is disclosed.

### A. Optimisation Modulo Theories

In the following, we briefly explain how to encode our problem in SMT. The goal is to compute satisfying assignments of $[\![l]\!]$ of minimal costs in $\mathcal{K}$. We achieve this objective by solving a sequence $\pi_0, \ldots, \pi_n$ of SMT problems, progressively tightening the bound on the cost of the solution we are looking for, according to the partial order $\sqsubseteq$. The cost of an assignment satisfying $[\![l]\!]$ is defined by the objective function

$$f(\overline{c_1}, \ldots, \overline{c_n}) = (\overline{c_1} \times \mathsf{cost}(c_1)) \oplus \cdots \oplus (\overline{c_n} \times \mathsf{cost}(c_n))$$

where $\overline{c_i} \times \mathsf{cost}(c_i)$ returns $\bot$ if $\overline{c_i} = \mathsf{ff}$, or $\mathsf{cost}(c_i)$ otherwise: we count the cost of guessing channel $c$ only if $c$ is needed to achieve the attack, i.e., when $\overline{c} = \mathsf{tt}$.
Problem $\pi_0$ corresponds to

$$[\![l]\!] \wedge (\mathsf{goal} := f(\overline{c_1}, \ldots, \overline{c_n}))$$

where the value of the solution (if any) is stored in variable goal. Moreover, the monoid has to be encoded in the SMT problem: numeric sets are already available in standard solvers, while more elaborate structures have to be explicitly formalised (elements, ordering relation, monoid operator).
Finally, we build a sequence $\pi_1, \ldots, \pi_n$ of problems, where $\pi_i$ is a more constrained version of $\pi_{i-1}$, in particular asking for $(i)$ a different solution and $(ii)$ a non-greater cost, until $\pi_n$ is reported unsatisfiable. The correctness of the algorithm stems from the construction of the sequence, according to which there cannot exist further assignments complying with the cost constraints after unsatisfiability has been claimed.
Finally, observe that even if SAT is NP-complete, modern solvers can handle problems with millions of variables. Moreover, this complexity is in line with the complexity of the minimisation analysis of [**?**], [3].

### B. Example

There are several techniques for quantifying the cost of guessing secret information. *Quantification of information leakage* [27] is an information theory-based approach for estimating the information an adversary gains about a given secret $s$ by observing the behaviour of a program parametrised on $s$. If $s$ is quantified in bits, then the corresponding information leaked by the program is quantified as the number of bits learnt by the adversary by observing one execution of the system. For instance, consider a test program $T$ parametrised on a secret password. $T$ inputs a string and answers whether or not the password is matched. Under the assumptions that the adversary knows the program and the length of the secret (no security-by-obscurity), we can estimate the knowledge gained by the adversary after one guessing attempt.
We leverage QUAIL [28], a freely-available tool for quantifying information leakage, for determining costs to channels. Denoted $\lambda_T(s)$ the leakage of $T$ on a secret $s$, we quantify the strength of a channel $c$ of $n$ bits as

$$\mathsf{cost}(c) = \frac{n}{\lambda_T(c)}$$

where we assume the security offered by $c$ to be uniformly distributed over the $n$ bits. In this settings we are thus working in the cost monoid $(\mathbb{Q}, +)$.
In our running example, the secrets to be guessed are pwd, otp, cert, pin, while we assume that id is known to the attacker and thus has cost 0 (in particular, in the *NemID* system is not difficult to retrieve such id, corresponding to the social security number of an individual). Moreover, we know that pwd contains between 6 and 40 alphanumeric symbols and it is not case sensitive: assuming an average length of 10 symbols, given that there are 36 such symbols, we need 5.17 bits to represent each symbol, for a total length of 52 bits.

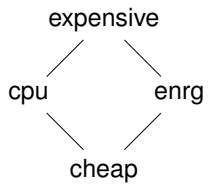



Figure 3. The Hasse diagram of a partially ordered cost structure.

Analogously, we determine the length of otp as 20 bits, while the length of the pin depends on the service provider: in case of a major bank it is just 14 bits. As for the certificate, the authority is following NIST recommendations, using 2048-bit RSA keys for the time being, and for the sake of simplicity we assume that guessing an RSA key cannot be faster than guessing each of the bits individually. Finally, we disregard login by assigning it the least upper bound of the costs of all the other channels. Exploiting QUAIL and the formula defined above, we obtain the following cost map:

$$\mathsf{cost}(\mathsf{pwd}) = 4.4 \times 10^{15} \quad \mathsf{cost}(\mathsf{pin}) = 1.5 \times 10^{4}$$
$$\mathsf{cost}(\mathsf{otp}) = 10^{6} \quad \mathsf{cost}(\mathsf{cert}) = 3.4 \times 10^{616}$$

Fed to the SMT-based optimisation engine, the problem is found to be satisfiable with cheapest assignment

$$[\mathsf{id} \mapsto \mathsf{tt}, \mathsf{pin} \mapsto \mathsf{tt}]$$

whose cost is $1.5 \times 10^4$ bits, meaning that the most practicable way to break the authentication protocol is attacking the mobile app, as long as we believe that our cost map is sensible.

We have shown one elegant way of quantifying the cost of guessing a channel in the monoid $(\mathbb{Q}, +)$, but any cost map suitable to a specific application can be used. Nonetheless, it is often difficult to provide an absolute estimate of the strength of a protection mechanism: sometimes different mechanisms are even incomparable, as cyber and physical means might be. In such cases, it is more natural to describe the relative strength of a set of mechanisms with respect to each other. This is achieved in the implementation by computing the analysis over symbolic and partially ordered cost structures. As a basic example, consider the lattice displayed in Fig. 3: we could characterise the cost of obtaining given information as cheap, if it does not require a specific effort, as cpu, if it requires significant computational capabilities (e.g. breaking an encryption scheme), as enrg, if it requires to spend a considerable amount of energy (e.g. engaging in the wireless exchange of a number of messages), or as expensive, if it requires both computations and energy. In order to combine such costs, a suitable choice is to take as monoid operator $\oplus$ the least upper bound $\sqcup$ of two elements in the cost lattice.

Observe that in general there could be more than one optimal model for $[\![l]\!]$, and our tool computes all of them.

## VII. The Quality Tree Generator

A proof-of-concept implementation of the framework has been developed in Java and is available at

http://www.imm.dtu.dk/~rvig/quality-trees.html

together with the code for the *NemID* example described in the paper.

The Quality Tree Generator takes as input an ASCII representation of a Value-Passing Quality Calculus process $P$ and generates the set $[\![P]\!]\mathsf{tt}$. Moreover, given a label $l$ occurring in $P$, the tool generates the formula $[\![l]\!]$, and given the cost to channels computes the cheapest assignments to $[\![l]\!]$.

Costs can be specified in two ways: numeric costs can be directly fed to the tool, while before specifying symbolic costs the finite lattice $(\mathcal{K}, \sqsubseteq)$ has to be loaded. In order to specify a lattice, one as to declare $\top$ and $\bot$, and then operator $\oplus$ as a list of entries $x \oplus y = z$. The names of the elements of the lattice and the partial order $\sqsubseteq$ are automatically inferred from the graph of $\oplus$.

As for the engine, we have implemented the backward-chaining procedure of Sect. V, defining our own simple infrastructure for propositional logic, as available libraries tend to avoid the explicit representation of implications, that is instead handy in our case during the backward-chaining computation.

Once the backward-chaining procedure is executed, and thus $[\![l]\!]$ has been derived, the tool can graphically represent the corresponding tree $T_l$, thanks to an encoding in DOT[2] and using ZGRViewer[3] for displaying it.

In order to compute attacks (i.e., assignments to $[\![l]\!]$) of optimal cost we rely on Microsoft Z3 [29] (Java API), an SMT solver. All these components are glued together thanks to a simple graphical interface.

## VIII. Conclusion

The increasing complexity of IT systems demands for a formal investigation of their security properties, able to quantify the threats to which they are subject and to treat cyber and physical features in a uniform manner. Attack trees have proven a useful tool to study threat scenarios and convey them in an intuitive way, but any manual construction is doomed to be incomplete whenever the size of the tree exceeds a few hundred nodes.

In order to tackle this problem, we have presented a novel method for the automated generation of attack trees. In particular, our technique improves on the existing literature by resorting to a process algebraic specification of the system. This choice allows to model a great many scenarios, beyond the standard network security domain, and enables designing syntax-directed static analyses, avoiding the systematic state space explosion suffered by model checking algorithms, even if retaining an exponential worst-case complexity. Moreover, process calculi have proven useful notations for the formal design of complex systems, embracing the need for analysing vulnerabilities at design time, and thus before the actual system is produced.

In our process-algebraic specification we identify the notion of attack as a set of channels that an adversary has to know in order to attain a given location in the system. Hence, our approach handles in a uniform way both cyber and physical protection mechanisms, such as channels exploiting cryptography, or reinforced gates. Moreover, if a map from channels to costs is provided, the framework computes the cheapest sets of channels that enable attaining a location of interest. Both numerical and partially-ordered symbolic cost structures can be relied on, so as to facilitate the encoding of the problem

[2] http://www.graphviz.org/

[3] http://zvtm.sourceforge.net/zgrviewer.html

when it is hard to devise sensible absolute estimates of the available security mechanisms.

The feasibility of the approach is witnessed by a freely-available implementation, and has been demonstrated on the study of a real system used for authentication purposes on a national scale.

As future work, besides consolidating the proof-of-concept implementation, we plan to investigate how the capability of testing data carried by input variables integrates in the framework. The basic ideas behind the necessary technical developments are sketched in Appendix B: it is however unclear to us whether the benefit in providing more detailed trees would balance the increase in complexity of their structure and therefore the drop in human-readability. Finally, a direct performance comparison with existing tools based on model checking would be interesting, even though the usefulness of our approach partly lies in the capability of dealing with scenarios not encodable in those tools.

## ACKNOWLEDGMENT

This work is supported by the IDEA4CPS project, granted by the Danish Research Foundations for Basic Research (DNRF86-10). Special thanks to Zaruhi Aslanyan and Alessandro Bruni for many inspiring and fruitful discussions.

## REFERENCES

[1] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 workshop on New security paradigms NSPW 98*, vol. pages, 1998, pp. 71–79.

[2] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," in *Proceedings 15th IEEE Computer Security Foundations Workshop CSFW15*, 2002, pp. 49–63.

[3] O. Sheyner and J. Wing, "Tools for Generating and Analyzing Attack Graphs," in *2nd International Symposium on Formal Methods for Components and Objects (FMCO'03)*, ser. LNCS, vol. 3188. Springer, 2004, pp. 344–371.

[4] V. Mehta, C. Bartzis, H. Zhu, E. Clarke, and J. Wing, "Ranking Attack Graphs," in *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, ser. LNCS, vol. 4219. Springer, 2006, pp. 127–144.

[5] B. Schneier, "Attack Trees," *Dr. Dobb's Journal*, 1999.

[6] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated Generation and Analysis of Attack Graphs," in *2002 IEEE Symposium on Security and Privacy*, 2002, pp. 273–284.

[7] S. Mauw and M. Oostdijk, "Foundations of Attack Trees," in *8th International Conference on Information Security and Cryptology (ICISC'05)*, ser. LNCS, vol. 3935. Springer, 2006, pp. 186–198.

[8] M. Rehák, E. Staab, V. Fusenig, M. Pěchouček, M. Grill, J. Stiborek, K. Bartoš, and T. Engel, "Runtime Monitoring and Dynamic Reconfiguration for Intrusion Detection Systems," in *Recent Advances in Intrusion Detection (RAID'09)*, ser. LNCS, vol. 5758. Springer, 2009, pp. 61–80.

[9] A. Jürgenson and J. Willemson, "Serial Model for Attack Tree Computations," in *Information, Security and Cryptology (ICISC'09)*, ser. LNCS, vol. 5984. Springer, 2010, pp. 118–128.

[10] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer, "Foundations of Attacks-Defense Trees," in *7th International Workshop on Formal Aspects of Security and Trust (FAST'10)*, ser. LNCS, vol. 6561. Springer, 2010, pp. 80–95.

[11] A. Roy, D. S. Kim, and K. S. Trivedi, "Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees," *Security and Communication Networks*, vol. 5, no. 8, pp. 929–943, 2012.

[12] R. W. Baldwin, "Rule Based Analysis of Computer Security," 1987. [Online]. Available: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-401.pdf

[13] M. Dacier, Y. Deswarte, and M. Kaaniche, "Models and tools for quantitative assessment of operational security," in *12th International Information Security Conference (IFIP/SEC'96)*, 1996, pp. 177–186.

[14] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS'02. ACM, 2002, pp. 217–224.

[15] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS'06. ACM, 2006, pp. 336–345.

[16] S. Bistarelli, M. Dall'Aglio, and P. Peretti, "Strategic games on defense trees," in *Formal Aspects in Security and Trust (FAST'06)*, ser. LNCS, vol. 4691. Springer, 2007, pp. 1–15.

[17] B. Kordy, S. Mauw, and P. Schweitzer, "Quantitative Questions on Attack-Defense Trees," in *15th International Conference on Information Security and Cryptology (ICISC'12)*, ser. LNCS, vol. 7839. Springer, 2012, pp. 49–64.

[18] B. Blanchet, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, 2009.

[19] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico, "Satisfiability Modulo the Theory of Costs: Foundations and Applications," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 6015, 2010, pp. 99–113.

[20] R. Vigo, F. Nielson, and H. Riis Nielson, "Uniform Protection for Multi-exposed Targets," in *E. Ábrahám and C. Palamidessi (Eds.), FORTE 2014*, ser. LNCS, vol. 8461. Springer, 2014, pp. 182–198.

[21] S. Mödersheim and L. Viganò, "Secure Pseudonymous Channels," in *14th European Symposium on Research in Computer Security (ESORICS'09)*, ser. LNCS, vol. 5789. Springer, 2009, pp. 337–354.

[22] H. R. Nielson, F. Nielson, and R. Vigo, "A Calculus for Quality," in *9th International Symposium on Formal Aspects of Component Software (FACS'12)*, ser. LNCS, vol. 7684. Springer, 2012, pp. 188–204.

[23] R. Vigo, F. Nielson, and H. R. Nielson, "Broadcast, Denial-of-Service, and Secure Communication," in *10th International Conference on integrated Formal Methods (iFM'13)*, ser. LNCS, vol. 7940, 2013, pp. 410–427.

[24] J. Mallios, S. Dritsas, B. Tsoumas, and D. Gritzalis, "Attack Modeling of SIP-Oriented SPIT," in *2nd International Workshop on Critical Information Infrastructures Security (CRITIS'07)*, ser. LNCS, vol. 5141, 2008, pp. 299–310.

[25] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice-Hall, 2009.

[26] C. Meadows, "A cost-based framework for analysis of denial of service in networks," *Journal of Computer Security*, vol. 9, no. 1, pp. 143–164, 2001.

[27] F. Biondi, A. Legay, P. Malacaria, and A. Wasowski, "Quantifying Information Leakage of Randomized Protocols," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, ser. LNCS, vol. 7737. Springer, 2013, pp. 68–87.

[28] F. Biondi, A. Legay, L.-M. Traonouez, and A. Wasowski, "QUAIL: A Quantitative Security Analyzer for Imperative Code," in *Computer Aided Verification (CAV'13*, ser. LNCS, vol. 8044. Springer, 2013, pp. 702–707.

[29] L. de Moura and N. Bjørner, "Z3 : An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, ser. LNCS, vol. 4963, 2008, pp. 337–340.

[30] F. Nielson, H. R. Nielson, and R. R. Hansen, "Validating firewalls using flow logics," *Theoretical Computer Science*, vol. 283, no. 2, pp. 381–418, 2002.

## APPENDIX A
## BROADCAST SEMANTICS

The labelled semantics of the Value-Passing Quality Calculus is presented in Table V, and is parametrised on the structural congruence relation of Table IV. In the congruence, we denote the free names in a process $P$ as $\mathsf{fn}(P)$, where $(\nu c)\, P$

Table IV. THE STRUCTURAL CONGRUENCE OF THE CALCULUS.

| | |
|---|---|
| $P \equiv P$ | $P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$ |
| $P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$ | $P \mid \mathsf{0} \equiv P$ |
| $P_1 \mid P_2 \equiv P_2 \mid P_1$ | $P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$ |
| $(\nu c)\, P \equiv P$ if $c \notin \mathsf{fn}(P)$ | $(\nu c_1)\,(\nu c_2)\, P \equiv (\nu c_2)\,(\nu c_1)\, P$ |
| $!P \equiv P \mid !P$ | $P_1 \equiv P_2 \Rightarrow C[P_1] \equiv C[P_2]$ |

$$(\nu c)\,(P_1 \mid P_2) \equiv ((\nu c)\, P_1) \mid P_2 \text{ if } c \notin \mathsf{fn}(P_2)$$

binds the name $c$ in process $P$. Moreover, the congruence holds for contexts $C$ defined as

$$C ::= [\,] \;\mid\; (\nu c)\, C \;\mid\; C \mid P \;\mid\; P \mid C$$

As usual in $\pi$-like calculi, processes are congruent under $\alpha$-conversion, and renaming is enforced whenever needed in order to avoid accidental capture of names during substitution.
The semantics of Table V is based on the transition relation $P \Longrightarrow P'$, which is enabled by combining a local transition $\xrightarrow{\lambda}$ with the structural congruence.
The semantics models asynchronous broadcast communication, and makes use of a label $\lambda ::= \tau \mid c_1!c_2$ to record whether or not a broadcast $c_1!c_2$ is available in the system. If not, label $\tau$ is used to denote a silent action. Output is thus a non-blocking action, and when it is performed the broadcast is recorded on the arrow. When a process guarded by a binder receives an output, the broadcast remains available to other processes willing to input. This behaviour is encoded in rules (In-ff) and (In-tt), where we distinguish the case in which a binder has not received enough input, and thus keeps waiting, from the case in which a binder is satisfied and thus the computation may proceed, applying the substitution induced by the received communication to the continuation process. These rules rely on two auxiliary relations, one defining how an output affects a binder, and one describing when a binder is satisfied (enough inputs have been received), displayed in the third and fourth section of Table V, respectively.
We write $c_1!c_2 \vdash b \rightarrow b'$ to denote that the availability of a broadcast makes binder $b$ evolve into binder $b'$. When a broadcast name $c_2$ is available on the channel over which an input is listening, the input variable $x$ is bound to the expression $\mathsf{some}(c_2)$, marking that something has been received. Otherwise, the input is left unchanged. Technically, the syntax of binders is extended to include substitutions. This behaviour is seamlessly embedded into quality binders: a single output can affect a number of sub-binders, due to the broadcast paradigm and to the intended semantics of the quality binder, according to which the sub-binders are *simultaneously* active.
As for evaluating whether or not enough inputs have been received, the relation $b ::_v \theta$ defines a boolean interpretation of binders. An input evaluates to $\mathsf{ff}$, for it must be performed before continuing with the computation; a substitution evaluates to $\mathsf{tt}$, since it stands for a received input; a quality binder is evaluated applying the quality guard, which is a boolean predicate, to the boolean values representing the status of the sub-binders. The substitution related to a quality binder is obtained by composing the substitutions given by its sub-binders.
It is worthwhile observing that substitutions are applied directly by the semantics, and as we consider closed processes, whenever an output $c_1!t$ is ready to execute, $t$ must have been replaced by a name $c_2$. For this reason the evaluation of a case clause is straightforward, as the variable $x$ checked by the instruction must have been replaced with a constant expression.
Finally, rules (Res-) and (Par-) take care of restrictions and interleaving. It is worthwhile observing that the transitions of these constructs cannot be embedded in contexts. On the one hand the labelled semantics, which does not mention the synchronising prefixes explicitly in the input rules, compels to consider the case of bound outputs; on the other hand, the broadcast paradigm forces us to prevent possibly synchronising processes to interleave. Therefore, a process $(\nu c)\, P$ is forbidden to use the name $c$ in a broadcast before having extruded the scope of $c$. In the case of parallel composition, if a process $P_1$ can make a broadcast, it is allowed to interleave with any process that is not ready to make an input, in which case either rule (In-ff) or (In-tt) applies. Moreover, the formalisation obliges to execute $\tau$-transitions before unfolding a replication or broadcasting a value.

Finally, observe that the distinction between local ($\xrightarrow{\lambda}$) and global transitions ($\Longrightarrow$), and the identification of the semantics with the latter, ensures that names are properly extruded as dictated by rule (Sys) in Table V, where $(\nu \overrightarrow{c})$ denotes the restriction of a list of names.

## APPENDIX B
## FIRST-ORDER ATTACK TREES

We present in this section an extension to the framework whose detailed development deserves to be deepened in future work. The ideas discussed in the following have not been implemented in the tool of Sect. VII.
The notion of knowledge needed to perform an attack adopted so far shifts the semantics load on the concept of secure channel. Besides its simplicity, this abstraction proves useful to model a great many different domains and lead to a sensible notion of attack tree. Nevertheless, it seems interesting to explore less abstract scenarios, where messages exchanged over channels enjoy a structure and their content is exploitable in the continuation. There is a substantial corpus of literature on how to extend a process calculus to handle reasoning on terms (e.g., via equational theories or pattern matching), but at the semantic heart of such calculi lies the capability of testing if what is received matches what was expected.
In order to fully encompass the original Quality Calculus we should introduce both testing capabilities and structured messages. Due to space constraints we show how to deal with the first extension, and we refrain from discussing the second one, which is obtained defining terms and expressions over an algebraic signature. As a matter of fact, distinguishing between a term $t$ and an expression $\mathsf{some}(t)$ we are already dealing with a (very simple) signature, and this gives the necessary insight onto our idea.
The syntax of the Value-Passing Quality Calculus, introduced in Sect. III, is enhanced as follows. First of all, we allow now input and output channels to range over terms $t$, writing $t?x$ and $t_1!t_2$. Secondly, we update the case clause as ${}^{l}\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(t) : P_1\ \mathsf{else}\ P_2$, allowing to check the data payload (if any) of an input variable $x$. The semantics of

Table V. THE TRANSITION RULES OF THE CALCULUS.

$$\frac{P_1 \equiv (\nu \overrightarrow{c})\, P_2 \quad (\nu \overrightarrow{c})\, P_2 \xrightarrow{\lambda} P_3}{P_1 \Longrightarrow P_3} \quad \text{(Sys)}$$

$${}^{l}c_1!c_2.P \xrightarrow{c_1!c_2} P \quad \text{(Out)}$$

$$\frac{P_1 \xrightarrow{c_1!c_2} P_1' \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{ff}} \theta}{P_1 \mid {}^{l}b.P_2 \xrightarrow{c_1!c_2} P_1' \mid {}^{l}b'.P_2} \quad \text{(In-ff)}$$

$$\frac{P_1 \xrightarrow{c_1!c_2} P_1' \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{tt}} \theta}{P_1 \mid {}^{l}b.P_2 \xrightarrow{c_1!c_2} P_1' \mid P_2\theta} \quad \text{(In-tt)}$$

$${}^{l}\mathsf{case\ some}(c)\ \mathsf{of\ some}(y) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1[c/y] \quad \text{(Then)}$$

$${}^{l}\mathsf{case\ none\ of\ some}(y) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2 \quad \text{(Else)}$$

$$\frac{P \xrightarrow{\tau} P'}{(\nu c)\, P \xrightarrow{\tau} (\nu c)\, P'} \quad \text{(Res-tau)}$$

$$\frac{P \xrightarrow{c_1!c_2} P'}{(\nu c)\, P \xrightarrow{c_1!c_2} (\nu c)\, P'}\ \mathsf{if}\ c \neq c_1 \wedge c \neq c_2 \quad \text{(Res-out)}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2} \quad \text{(Par-tau)}$$

$$\frac{P_1 \xrightarrow{c_1!c_2} P_1'}{P_1|P_2 \xrightarrow{c_1!c_2} P_1'|P_2}\ \mathsf{if}\ P_2 = !P_2' \vee P_2 = {}^{l}c_1'!c_2'P_2' \quad \text{(Par-out)}$$

$$c_1!c_2 \vdash c_1?x \to [\mathsf{some}(c_2)/x]$$

$$c_1!c_2 \vdash c_3?x \to c_3?x \quad \mathsf{if}\ c_1 \neq c_3$$

$$\frac{c_1!c_2 \vdash b_1 \to b_1' \quad \cdots \quad c_1!c_2 \vdash b_n \to b_n'}{c_1!c_2 \vdash \&_q(b_1, \ldots, b_n) \to \&_q(b_1', \ldots, b_n')}$$

$$c?x ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad [\mathsf{some}(c)/x] ::_{\mathsf{tt}} [\mathsf{some}(c)/x]$$

$$\frac{b_1 ::_{v_1} \theta_1 \quad \cdots \quad b_n ::_{v_n} \theta_n}{\&_q(b_1, \ldots, b_n) ::_v \theta_n \cdots \theta_1}\ [\![q]\!](v_1, \ldots, v_n) = v$$

Appendix A is modified accordingly:

$${}^{l}\mathsf{case\ some}(c)\ \mathsf{of\ some}(c) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1$$
$${}^{l}\mathsf{case\ some}(c)\ \mathsf{of\ some}(y) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1[c/y]$$
$${}^{l}\mathsf{case\ some}(c)\ \mathsf{of\ some}(c') \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2\ \mathsf{if}\ c \neq c'$$
$${}^{l}\mathsf{case\ none\ of\ some}(c) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2$$
$${}^{l}\mathsf{case\ none\ of\ some}(y) \colon P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2$$

The translation from processes to formulae of Sect. IV is lifted from propositional to first-order logic, so as to account for the richer expressiveness of the case clause:

$$\begin{aligned}[\![{}^{l}\mathsf{case}\ x\ \mathsf{of\ some}(t) \colon P_1\ \mathsf{else}\ P_2]\!]\Phi = \\ [\![P_1]\!](\Phi \wedge \exists \mathsf{fv}(t).(x = \mathsf{some}(t)) \cup \\ [\![P_2]\!](\Phi \wedge \neg(\exists \mathsf{fv}(t).(x = \mathsf{some}(t))) \cup \\ \{\Phi \Rightarrow \bar{l}\}\end{aligned}$$

where $\mathsf{some}(\cdot)$ is a unary predicate, $\mathsf{fv}(t)$ denotes the variables free in $t$, and we write $x$ instead of $\overline{x}$ for now $x$ ranges over a set of optional data. Similarly, the translation of binders has now to record the term to which an input variable is bound when the corresponding binder is satisfied:

$$\mathsf{th}(\Phi, t?x) = \{\exists y.(\Phi \wedge t \Rightarrow (x = \mathsf{some}(y))\}$$

where $t$ ranges over a set of data (the translation of output has to be updated similarly).
Finally, for building the tree some unification is needed in the backward-chaining search of Sect. V:

$$[\![\exists \mathsf{fv}(t)(x = \mathsf{some}(t))]\!]\mathcal{D} = \bigvee_{\{h \mid \exists \mathsf{fv}(t')(h \Rightarrow (x = \mathsf{some}(t'))) \in [\![P]\!]\mathsf{tt}\ \wedge\ \exists \sigma.t = t'\sigma\}} [\![h\sigma]\!]\mathcal{D}$$

$$[\![\neg\exists \mathsf{fv}(t)(x = \mathsf{some}(t))]\!]\mathcal{D} = \bigwedge_{\{h \mid \exists \mathsf{fv}(t')(h \Rightarrow (x = \mathsf{some}(t'))) \in [\![P]\!]\mathsf{tt}\ \wedge\ \exists \sigma.t = t'\sigma\}} [\![\neg h\sigma]\!]\mathcal{D}$$

where $\sigma$ is a most general unifier.
We have thus shown how to lift all levels of the framework to name-passing calculi with standard testing capabilities. From a high-level perspective, the extension allows inspecting how security checks are performed, while the developments in the paper consider checks as atomic entities, distinguishing between them through the cost map.
Though such an extension may sound interesting to the scientist, it is unclear to us whether the more detailed "first-order" trees would be of any use in practice, the main risk being that additional information would decrease readability drastically. In addition to this, whenever a finer-grained investigation is needed, we could take advantage of the modularity of the propositional framework, as discussed in Sect. IV-B.
Finally, in order to carry the extension to the Quality Tree Generator of Sect. VII, the main obstacle would be to introduce unification of terms in the backward-chaining procedure. While the tool has been implemented in Java for easing the development of a graphical interface, the extension naturally calls for a functional approach.

## APPENDIX C
## CORRECTNESS OF ATTACK TREES

### A. Property of $[\![P]\!]\mathsf{tt}$

*Proof of Lemma 1:* By induction on the structure of processes. In particular, observe that in Table II a literal $\overline{x}$ is added to $\Phi$ only when a case clause is met, and by hypothesis $\overline{x}$ must previously appear in a binder, for processes are closed. ∎

Let us discuss now the *complexity* of the translation given in Table II. Let $\mathsf{size}(\mathcal{C})$ denote the number of literals occurring in a set of formulae $\mathcal{C}$, that is, $\mathsf{size}(\mathcal{C}) = \sum_{\varphi \in \mathcal{C}} (\mathsf{size}(\varphi))$, where $\mathsf{size}(\varphi)$ counts the literals in $\varphi$.

**Lemma 2.** *Let $P$ be a closed process in the Value-Passing Quality Calculus. Assuming that $P$ contains $n$ actions, then* $\mathsf{size}([\![P]\!]\mathtt{tt}) = O(n^2)$.

*Proof:* If $P$ consists of $n$ actions, $[\![P]\!]\mathtt{tt}$ consists of at most $O(n)$ formulae[4]. The number of literals in a formula depends linearly on the number actions preceding the label at which the formula is generated (cf. Table II), hence the number of literals in $[\![P]\!]\mathtt{tt}$ is asymptotically bounded by $n^2$. ∎

It is interesting to observe that from a theoretical point of view $O(n^2)$ is a precise bound to $\mathsf{size}([\![P]\!]\mathtt{tt})$. Consider the process $IN_n$ that consists of $n$ sequential inputs $c_1?x_1.\ldots.c_n?x_n$. The number of literals in $[\![IN_n]\!]\mathtt{tt}$ grows with

$$\begin{aligned}\textstyle\sum_{i=1}^{n}(2(i-1)+3) &= \textstyle\sum_{i=1}^{n}(2i+1) = \\ &= n + 2\textstyle\sum_{i=1}^{n} i = n + 2\frac{n(n+1)}{2} = \\ &= n^2 + 2n\end{aligned}$$

where $i$ records the number of literals in the hypothesis $\Phi$, we have omitted counting the $\mathtt{tt}$ conjuncts, and we leverage the fact that an input generates two formulae whose size is $\mathsf{size}(\Phi)+2$ adding 1 literal to the hypothesis, from which the relation $2(i-1)+3$ is derived. Similarly, the translation of a process made of alternating inputs and $\mathsf{case}$ clauses would grow quadratically (with greater constants than $IN_n$).

### B. Correctness of $[\![l]\!]$

The correctness statement supporting the procedure of Table III and validating the solution technique of Sect. VI is stated in Th. 1 below.

Before presenting the result, let us fix some notation. Given a truth assignment $m$ and a propositional formula $\varphi$, we say that $m$ satisfies $\varphi$ (or, equivalently, $m$ is model for $\varphi$), denoted $m \models \varphi$, if the values assigned by $m$ to the variables in $\varphi$ are such that $\varphi$ evaluates to $\mathtt{tt}$. The value of a propositional variable $v$ in $m$ is denoted $m(v)$. Moreover, given a truth assignment $m$ and a set of inference rules $\mathcal{R}$, we say that a formula $\varphi$ is derived from $m$ by $\mathcal{R}$, denoted $m \vdash_{\mathcal{R}} \varphi$, if there is a proof showing that $\varphi$ is derived by $m$ under the inference rules $\mathcal{R}$ together with classic propositional rules.

**Theorem 1.** *Let $P$ be a closed process in the Value-Passing Quality Calculus, and assume that each variable $x$ and name $c$ occurring in $P$ is bound exactly once. Consider the translation* $[\![P]\!]\mathtt{tt}$ *into propositional formulae; then, for each label $l$ occurring in $P$ and truth assignment $m$, it holds*

$$m \models [\![l]\!] \;\Rightarrow\; m \vdash_{[\![P]\!]\mathtt{tt}} \bar{l}$$

*Proof sketch.* Let $\varphi$ be the antecedent in $\varphi \Rightarrow \bar{l}$, which exists and is unique in $[\![P]\!]\mathtt{tt}$. Since $[\![l]\!] = [\![\varphi]\!]\emptyset$ and $\varphi \Rightarrow \bar{l} \in [\![P]\!]\mathtt{tt}$, it suffices to show

$$m \models [\![\varphi]\!]\emptyset \;\Rightarrow\; m \vdash_{[\![P]\!]\mathtt{tt}} \varphi$$

from which the statement follows. The proof is by induction on the number of steps in the unfolding of the generation of $[\![\varphi]\!]\emptyset$, and relies on the validity of the rules in Table III as inference rules.

**Theorem 2.** *Let $P$ be a closed process in the Value-Passing Quality Calculus, and assume that each variable $x$ and name $c$ occurring in $P$ is bound exactly once. Consider the translation* $[\![P]\!]\mathtt{tt}$ *of $P$ into propositional formulae. Then, for all labels $l$ occurring in $P$, it holds*

$$\textit{if } P|Q \Longrightarrow^{*} C[{}^{l}P'] \textit{ then}$$
$$\exists m \textit{ s.t. } m \models [\![l]\!] \;\wedge\; \{\bar{c} \,|\, m(\bar{c}) = \mathtt{tt}\} \subseteq \mathsf{fn}(Q)$$

*Proof sketch.* Technically, it seems convenient to organise a formal proof in two lemmata. First, if $P|Q$ reaches $l$ then $P|H[\mathsf{fn}(Q)]$ reaches $l$, where process $H$ is the hardest attacker possible and is parametrised on the knowledge of $Q$. $H$ can be thought of as the process executing all possible actions on $\mathsf{fn}(Q)$, and the proof simply argues that whatever $Q$ can, $H$ can. A similar proof is detailed in [30].

The second lemma concludes showing that if $P|H[M]$ reaches $l$, then there must be a model $m$ of $[\![l]\!]$ such that $\{\bar{c} \,|\, m(\bar{c}) = \mathtt{tt}\} \subseteq M$. The proof combines induction on the length of the derivation sequence leading to $l$ with Th. 1. In the interesting case of quality binders, the computation of $[\![l]\!]$ ensures that all the minimal combinations are considered that allow passing a binder, and thus also the one chosen by $H$.

**Lemma 3.** *Let $P$ be a closed process in the Value-Passing Quality Calculus, and assume that each variable $x$ and name $c$ occurring in $P$ is bound exactly once. Then, for all labels $l$ occurring in $P$, the formula $[\![l]\!]$ built according to the rules of Table III contains no literal $\overline{x}$. In particular, $[\![l]\!]$ only contains literals related to channels $c$.*

*Proof:* By induction on the number of steps in the unfolding of the generation of $[\![l]\!]$, according to the rules in Table III. ∎

[4]More in detail, $[\![P]\!]\mathtt{tt}$ consists of $n_o + n_c + n_i + n_b$ formulae, $n_o$ being the number of outputs in $P$, $n_c$ the number of $\mathsf{case}$ clauses, $n_i$ the number of simple inputs (including the ones occurring within quality binders), and $n_b$ the number of binders.