

Giampaolo Bella, Stefano Bistarelli,  
Simon N. Foley and Barry O'Sullivan (Eds.)

---

# Applications of Constraint Satisfaction and Programming to Computer Security Problems

First International Workshop  
Sitges, Spain, 1st October 2005  
Proceedings

Held in conjunction with the  
Eleventh International Conference on  
Principles and Practice of  
Constraint Programming (CP 2005)



# Preface

Constraint satisfaction and programming is emerging as an effective practical approach for solving large complex problems. It offers a significant body of successful techniques for verifying system properties. Recently, researchers have begun using advances in constraint programming and solving to solve security problems, with success. This workshop seeks to act as a catalyst for this emerging area by exploring the challenges and the potential that these techniques may offer when applied to security problems.

The use of constraint satisfaction and programming to address security problems is recent, and it has already produced a number of novel solutions and insights. For example, constraints have been successfully used in the the analysis of security protocols, the development of access control models and mechanisms, firewall configuration and secure system configuration in general.

The workshop aim was to provide a forum where researchers working in the area of security and constraints could discuss their most recent ideas and developments and think together about the most promising new directions. Therefore, we encouraged the presentation of work in progress or on specialized aspects of the area. Papers that bridge the gap between theory and practice were especially welcome.

This volume contains five contributed papers, the abstract of an invited talk given by Yannick Chevalier, and the abstract of a demonstration given by Fred Spiessens, Yves Jaradin, and Peter Van Roy.

We wish to thank all the authors who submitted papers and demonstrations to this workshop, the members of the programme committee, the invited speaker, and the CP-2005 Tutorial and Workshop Chairs, Alan Frisch and Ian Miguel. We would like to especially acknowledge the financial sponsorship received from the Istituto di Informatica e Telematica of CNR (Pisa, Italy) to support our invited speaker.

August 2005

Giampaolo Bella  
Stefano Bistarelli  
Simon N. Foley  
Barry O'Sullivan  
Programme Chairs

## Organising Committee

Giampaolo Bella – Università di Catania, Italy  
Stefano Bistarelli – Università degli studi "G. D'Annunzio" di Chieti-Pescara  
and IIT-CNR, Italy  
Simon N. Foley – University College Cork, Ireland  
Barry O'Sullivan – Cork Constraint Computation Centre, Ireland

## Programme Committee

Giampaolo Bella – Università di Catania, Italy  
Stefano Bistarelli – Università degli Studi "G. d'Annunzio" di Chieti-Pescara,  
and IIT-CNR, Italy  
Yannick Chevalier – Université Paul Sabatier, France  
Giorgio Delzanno – Università di Genova, Italy  
Alessandra Di Pierro – University of Pisa, Italy  
Fabio Fioravanti – Università degli Studi "G. D'annunzio", Pescara, Italy  
Simon Foley – University College Cork, Ireland  
John Herbert – University College Cork, Ireland  
Fabio Martinelli – Istituto di Informatica e Telematica, CNR, Pisa, Italy  
Barry O'Sullivan – Cork Constraint Computation Centre, Ireland  
Justin Pearson – Uppsala University, Sweden  
Michael Rusinowitch – INRIA Lorraine, France  
Vitaly Shmatikov – University of Texas at Austin, USA  
Fred Spiessens – U.C.L. Louvain-la-Neuve, Belgium  
Garret Swart – IBM Research, USA  
Peter Van Roy – Catholic University of Louvain, Belgium  
Luca Vigano – ETH, Zurich, Switzerland  
Duminda Wijesekera – George Mason University, USA  
Herbert Wiklicky – Imperial College London, UK

## Sponsorship

Istituto di Informatica e Telematica, CNR Pisa, Italy.

# Table of Contents

---

## I Invited Paper

---

From Cryptographic Protocol Analysis to Constraint Solving . . . . .	1
<i>Yannick Chevalier</i>	

---

## II Demonstration

---

Demo: A CCP based Tool to Analyze Patterns of Authority Propagation and Confinement . . . . .	2
<i>Fred Spiessens, Yves Jaradin, and Peter Van Roy</i>	

---

## III Contributed Papers

---

PS-LTL for Constraint-based Security Protocol Analysis . . . . .	3
<i>Ricardo Corin, Ari Saptawijaya, and Sandro Etalle</i>	
Masquerade Detection Using IA Network . . . . .	18
<i>Subrat Kumar Dash, Sanjay Rawat, G. Vijaya Kumari, and Arun K. Pujari</i>	
Distributed CLP Clusters as a Security Policy . . . . .	31
<i>Saket Kaushik, Duminda Wijesekera, William Winsborough, Paul Ammann</i>	
Heuristics for enforcing security constraints . . . . .	46
<i>Flemming Nielson and Hanne Riis Nielson</i>	
Using Constraints To Analyze And Generate Safe Capability Patterns . . .	61
<i>Fred Spiessens, Yves Jaradin, and Peter Van Roy</i>	



## From Cryptographic Protocol Analysis to Constraint Solving

Yannick Chevalier  
ychevali@irit.fr

Université Paul Sabatier Toulouse 3  
Toulouse, France

**Abstract.** The security of distributed systems such as *e-commerce*, virtual private networks, *e-administration*... heavily depends on the possibility of securely transferring data over an insecure medium. This is the role of cryptographic protocols, which have been intensively studied during the past years. This line of research was fruitful since, during the AVISPA project, several flaws were detected by an automated analysis of industrial-scale protocols [1].

In particular we will consider the Human Equivalent Privacy properties of authentication and secrecy. We will first present how the search on a violation of these properties can be reduced to the satisfiability of some specially constructed reachability problems. After having presented the most important results in this area, we will present a result obtained in a joint work with M. Rusinowitch [2] on the combination of such problems and the application to the decidability and complexity of protocol analysis.

We will also discuss the shortcomings of this approach with respect to cryptographically sound proofs, and present the AVISPA tool for automated protocol analysis in the case of perfect cryptography assumption.

### Acknowledgement

This work was partially supported by the CNRS ACI Jeunes Chercheurs JC 9005 and the Shared Cost RTD (FET open) AVISPA<sup>1</sup> project funded by the Information Society Technologies programme of the European Commission, as IST-2001-39252.

### References

1. Alessandro Armando, David Basin, Yohann Boichut, Yannick Chevalier, and al Et. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications . In Etessami Kousha and Rajamani Sriram, editors, *Computer-Aided Verification , Edinburgh, Scotland, UK*, pages 1–5. Springer-Verlag, 06-10 juillet 2005.
2. Y. Chevalier and M. Rusinowitch. Combining Intruder Theories. In L. Caires, G. F. Italiano, and L. et al. Monteiro, editors, *Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005*, pages 639–651, Lisbon, Portugal, July 2005.

---

<sup>1</sup> <http://www.avispa-project.org/>

## Demo: A CCP based Tool to Analyze Patterns of Authority Propagation and Confinement

Fred Spiessens, Yves Jaradin, and Peter Van Roy

Université catholique de Louvain  
Louvain-la-Neuve, Belgium  
{fsp,yjaradin,pvr}@info.ucl.ac.be

**Abstract.** We demonstrate a CCP based tool for analyzing authority propagation and confinement in patterns of collaborating entities. Such analysis is important because it reveals the boundary conditions in which the pattern is safe to use. The domain specific language "SCOLL" [JSV05] (Safe Collaboration Language) will be introduced and used to model the restricted behavior of the trusted subjects in a pattern. The model is concerned only with the authority-propagating aspects of behavior: in what circumstances will the trusted subject collaborate with another subject, will it accept authority propagated during this collaboration, and what authority (if any) will it provide to the collaborating subject. The model itself is an expressive extension of classical Take-Grant systems.

We first show how the tool calculates a safe but precise approximation to the maximal authority propagation in a pattern. Then we show the more powerful way of using the tool: to calculate maximally collaborative behavior for any trusted subject in a pattern, while respecting a given set of requirements on the global confinement of authority. We present some interesting patterns of collaboration and calculate and discuss the maximal behavior they allow for their key role subjects.

We discuss the future extension of the tool with semi-automatic derivation of behavior from code, providing a domain specific alternative for abstract interpretation and a developer aid for secure programming. Modeling collaborative behavior based on real code makes sense if the code is written in a capability secure language, because then authority can only become available through collaborative propagation (delegation).

### Acknowledgments

This work was partially funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935, and by the MILOS project of the Walloon Region of Belgium under convention 114856.

### References

- [JSV05] Yves Jaradin, Fred Spiessens, and Peter Van Roy. SCOLL : A Language for Safe Capability Based Collaboration *Paper submitted to POPL'06.*, 2005. Available at: <http://www.info.ucl.ac.be/~yjaradin/SCOLL.pdf>.



# **$\mathcal{PS}$ -LTL for Constraint-based Security Protocol Analysis**

Ricardo Corin, Ari Saptawijaya, and Sandro Etalle

Department of Computer Science, University of Twente, The Netherlands

**Abstract.** We introduce  $\mathcal{PS}$ -LTL (pronounced *pastel*), a pure-past security linear temporal logic.  $\mathcal{PS}$ -LTL allows one to specify several security properties including data freshness and different notions of authentication and secrecy. We give semantics and further propose a procedure to decide validity of properties for finite (albeit symbolic) execution traces, for a relevant subset of  $\mathcal{PS}$ -LTL (which includes all the mentioned properties). Our procedure is easily integrated with constraint-based protocol analysis approaches. We also provide a Prolog implementation.

## **1 Introduction**

Despite their apparent simplicity, the correct design of security protocols is known to be a challenging task, and tools based on formal methods (e.g. [4, 21, 20, 18, 12]) are often used to debug and verify protocols before their deployment. Most of these tools are state-based: an abstract version of the protocol is run in presence of the so-called Dolev-Yao [14] intruder who has complete control over the network. By exhaustive search, one can establish whether the protocol is flawed or not. In addition, tools based on *forward search* [23, 18, 7, 12] (as opposed to those employing *backward search*, like [20, 3]) can be regarded as true engineering tools, as they can be used for simulating, rapid prototyping and debugging of security protocols. Here, limiting the search space explosion due to the intruder's behaviour is a central issue, which can be dealt with elegantly and effectively by using *constraint solving*: instead of generating all possible traces of the systems, the tool produces only a finite number of *symbolic traces*, in which *constrained variables* are used in place of the messages generated by the intruder. Each symbolic trace stands for a (possibly infinite) set of concrete traces. Originally proposed by Millen and Shmatikov [23], this now popular method was later improved and extended in [7, 1, 11].

Unfortunately, constraint-based verification systems are poor when it comes to specifying the properties one wants to check. For example, checking authentication is done in an ad-hoc manner, by e.g. adding a protocol participant but not its corresponding party, and observing whether the participant can still finish its run. This is coarse-grained, and cumbersome to implement (besides this, only a built-in notion of authentication is implemented by Millen [22] in his Prolog implementation). Checking secrecy is also ad-hoc, by adding an artificial protocol role which expects a secret no other participant would send. As we shall see, when secrecy is an atomic predicate being part of a language, more interesting properties can be stated about secrecy.

*Contributions* We propose a language to specify security properties, based on linear temporal logic (LTL) with pure-past operators. As we shall see,  $\mathcal{PS}$ -LTL provides flexibility, allowing one to specify several security properties like authentication ([18, 10])

(including *aliveness*, *weak agreement* and *non-injective agreement*), secrecy (*standard secrecy* [2] and *perfect forward secrecy* [13]) and also data freshness. The semantics of  $\mathcal{PS}$ -LTL is defined on concrete (variable-free) traces, but we introduce a decision procedure which allows one to check a relevant subset of  $\mathcal{PS}$ -LTL on the *symbolic traces* produced by our tool [7]. This fragment is expressive enough to cover all the considered security properties. We show that the decision procedure is correct wrt the concretization function, which maps a symbolic trace to all the valid traces it represents. We have incorporated a  $\mathcal{PS}$ -LTL interpreter into our protocol verification tool [7] thereby providing a full verification system (an online demo is available [8]). The Prolog code together with the proofs that did not fit in this version can be found in an extended version of this paper [9].

To the best of our knowledge, in the context of constraint-based methods, this is the first language for the specification of security properties which is equipped with a sound procedure for evaluating formulas against a symbolic trace.

*Related Work* In previous work [6] we study *local* security properties.  $\mathcal{PS}$ -LTL provides more powerful temporal operators (e.g. the *yesterday* Y and *since* S operators), which in turn allows the writing of more expressive security properties. Besides that work,  $\mathcal{PS}$ -LTL is inspired by the successful and elegant NPATRL logic [24]. As shown in subsequent work [21], NPATRL is strictly less powerful than LTL; also in that paper it is mentioned that the implementation of NPATRL to NRL Protocol Analyzer [20] presents difficulties (e.g. the inability to mention several `learn`'s in the same formula, a restriction we do not impose). Further examination is required to better compare NPATRL and our logic.

Our treatment of pure-past LTL is an adaptation of Havelund and Rosu [16]. We provide a different semantics, tailored for security and constraint solving, but also include a different definition for the temporal operator *historically* H, which we believe preserves better the faithfulness to standard LTL.

Finally, we use special flags like *run* and *end* that indicate an agent is running and completing the protocol, respectively. They are useful to later specify authentication properties as correspondence assertions à la Gordon and Jeffrey [15].

## 2 Preliminaries

In this section we introduce the basic elements we need in the rest of the paper. We first introduce our term algebra and intruder deduction rules, then the protocol model.

*Term Algebra* Messages are represented as terms in a free algebra generated by the operators in Table 1 (left), from a set of variables  $\mathcal{V}$  (denoted by uppercase letters  $A, B, Na, K, \dots$ ), and a set of constants  $\mathcal{C}$  (denoted by lowercase  $a, b, na, k, \dots$ ), representing the agent identities, nonces (ie. random values) and keys. We use a special constant  $e \in \mathcal{C}$  to denote the intruder's identity. We have pairing, public keys and (symmetric/asymmetric) encryption. We assume that private keys are never part of messages, and so they are never leaked. The set of ground terms is denoted by  $\mathcal{T}^+$ . When  $t \in \mathcal{T}^+$ , we say that  $t$  is *ground*, otherwise it is *non-ground*. Substitutions (denoted by  $\sigma, \rho, \dots$ ) are finite mappings from  $\mathcal{V}$  to  $\mathcal{T}$ ; Ground substitutions map  $\mathcal{V}$  to  $\mathcal{T}^+$ . Given  $v \in \mathcal{V}$  and  $t \in \mathcal{T}$ ,  $[t/v]$  denotes the singleton substitution mapping of  $v$  to  $t$ . The variables of a term  $t$  are denoted as  $var(t)$ . A term  $t'$  is an *instance* of another

term  $t$  if there is a substitution  $\sigma$  s.t.  $t' = t\sigma$ . The same terminology is used for the (later introduced) events, protocol roles and traces.

$t_1, t_2 ::= c$	constant in $\mathcal{C}$	$\{t_1, t_2\} \rightarrow_{pair} \{(t_1, t_2)\}$
$v$	variable in $\mathcal{V}$	$\{(t_1, t_2)\} \rightarrow_{first} \{t_1\}$
$pk(t_1)$	public key	$\{(t_1, t_2)\} \rightarrow_{second} \{t_2\}$
$(t_1, t_2)$	pair	$\{t\} \rightarrow_{hash} \{h(t)\}$
$h(t_1)$	hash	$\{t_1, t_2\} \rightarrow_{senc} \{\{t_1\}_{t_2}\}$
$\{t_1\}_{t_2}$	symmetric encryption	$\{\{t_1\}_{t_2}, t_2\} \rightarrow_{sdec} \{t_1\}$
$\{t_1\}_{\vec{t_2}}$	asymmetric encryption	$\{t_1, t_2\} \rightarrow_{penc} \{\{t_1\}_{\vec{t_2}}\}$
		$\{\{t_1\}_{\vec{pk(e)}}\} \rightarrow_{pdec} \{t_1\}$

**Table 1.** Left: Grammar for terms. Right: DY rules.

*DY Rules* Rules are used to represent the abilities of the intruder, in the style of Dolev-Yao. Let  $A$  and  $B$  be two sets of terms, and let  $\ell$  be a rule label, representing the name of the rule. A *rule* is denoted by  $A \rightarrow_{\ell} B$ . We work with the set of rules given in Table 1 (right). As usual, the attacker is allowed to pair and split terms, hash, symmetrically encrypt terms with any (possibly non-atomic) key and decrypt symmetrically if the key is known to the attacker. Public-key encryption (*penc*) is modelled by allowing to encrypt with any key. Then, rule *pdec* models asymmetric decryption of a term encrypted with the attacker's public key. The attacker cannot decrypt any term encrypted with a different public key than his own, since we assume that private keys are not leaked.

We now define  $\mathcal{F}(T)$ , representing the terms the intruder can generate from the term set  $T$ :

**Definition 1.** Let  $T$  be a ground term set, and let  $f_{\ell}$  be a set operator defined as  $f_{\ell}(T) = T \cup B$  when  $A \subseteq T$  and  $A \rightarrow_{\ell} B$  is a DY rule. Then,  $\mathcal{F}(T)$  denotes the smallest set that contains  $T$  and is closed wrt  $f_{\ell}$  for all rule  $\ell$ .

*Protocol Model* Our protocol model is related to the strand-space formalism [25], although we sometimes use a different terminology, e.g. we call system scenario what in strand-spaces is called a *semibundle*. In the following, we introduce events, traces, protocol roles, and system scenarios.

**Definition 2.** An event is one of the following:

- A *communication event*: a pair  $\langle a : m \diamond b \rangle$  where  $a, b$  are variables or agent constants,  $\diamond \in \{\triangleleft, \triangleright\}$  and  $m$  is a term.  $a$  is called the *active party*, and  $b$  is the *passive party*. The event  $\langle a : m \triangleright b \rangle$  reads as “agent  $a$  sends message  $m$  with intended destination  $b$ ”. Symmetrically,  $\langle a : m \triangleleft b \rangle$  stands for “agent  $a$  receives message  $m$  apparently from  $b$ ”.
- A *status event*:  $p(D_1, \dots, D_n)$ , with  $D_i$  a term for  $i \in [1..n]$  and  $p$  is a function symbol.

*Example 3.* The following are examples of communication and status events:

- $\langle a : (a, na) \triangleright B \rangle$  is a communication event in which agent  $a$  sends pair  $(a, na)$  to another agent  $B$ . Being  $B$  a variable (rather than a constant  $b$ ), this event represents any concrete communication event  $\langle a : (a, na) \triangleright c \rangle$  for any agent  $c$ . In other words,  $B$  is still undetermined.
- $\langle b : (A, N_A) \triangleleft A \rangle$  is a communication event in which agent  $b$  receives the pair  $(A, N_A)$  from another agent  $A$ . Similarly, because of the presence of variables  $A$  and  $N_A$ , this stands for any concrete event  $\langle b : (c, n) \triangleleft c \rangle$  for any agent  $c$  and any term  $n$ .
- $run(a, B, initiator, na, k_{lt}, K_{sk})$  is a status event, used typically as a flag to indicate that agent  $a$  has been running a protocol as an initiator apparently with (a still unspecified)  $B$  agreeing on some data  $na$ ,  $k_{lt}$ , and  $K_{sk}$ . The meaning of variables are similar as in previous examples. Similarly, we use an *end* status event to indicate an agent finishing an execution.  $\square$

A *trace* is a finite sequence of events, with the empty trace denoted as  $\langle \rangle$ . Appending an event  $ev$  to trace  $tr$  is written  $\langle tr \ ev \rangle$ . Functions *last* and *length* have the usual meaning:  $last(\langle tr \ ev \rangle) = ev$  (*last* is undefined for the empty trace),  $length(\langle \rangle) = 0$  and  $length(\langle tr \ ev \rangle) = length(tr) + 1$ . The prefix trace consisting of the first  $i$  events is denoted as  $tr_i$ , with  $tr_0 = \langle \rangle$  and  $tr_m = tr$  for  $m \geq length(tr)$ .

**Definition 4.** A protocol role is a trace in which all events share the same active agent.

Given a protocol written in standard ‘ $A \rightarrow B : M$ ’ notation, it is straightforward to obtain its parametric protocol roles, as shown in the next example.

*Example 5.* Consider the BAN Concrete Andrew Secure RPC protocol [4], with the last message stripped out, since it is not necessary for security.

1.  $a \rightarrow b : (a, na)$
2.  $b \rightarrow a : \{(na, k_{st})\}_{k_{lt}}$
3.  $a \rightarrow b : \{na\}_{k_{st}}$

First  $a$  sends a message with her identity and a fresh nonce  $na$ . Upon receipt,  $b$  generates a short term session key  $k_{st}$ , encrypts it along with  $a$ ’s nonce  $na$  using the long-term key  $k_{lt}$ , shared previously with  $a$ . Finally,  $a$  replies with her nonce  $na$  encrypted with the newly established key  $k_{st}$ . In the following, we name protocol roles such as *init*, and *resp*, denoting an initiator and a responder respectively. The parametric protocol roles are then:

$$\begin{aligned}
 init(A, B, N_A, K_{lt}, K_{st}) &= \langle \langle A : (A, N_A) \triangleright B \rangle \langle A : \{(N_A, K_{st})\}_{K_{lt}} \triangleleft B \rangle \\
 &\quad run(A, B, initiator, N_A, K_{lt}, K_{st}) \langle A : \{N_A\}_{K_{st}} \triangleright B \rangle \\
 &\quad end(A, B, initiator, N_A, K_{lt}, K_{st}) \rangle \\
 resp(A, B, N_A, K_{lt}, K_{st}) &= \langle \langle B : (A, N_A) \triangleleft A \rangle run(B, A, responder, N_A, K_{lt}, K_{st}) \\
 &\quad \langle B : \{(N_A, K_{st})\}_{K_{lt}} \triangleright A \rangle \langle B : \{N_A\}_{K_{st}} \triangleleft A \rangle \\
 &\quad end(B, A, responder, N_A, K_{lt}, K_{st}) \rangle \quad \square
 \end{aligned}$$

The instantiation of a parametric protocol role gives another protocol role. The next step consists of gathering several protocol roles together, which provides a particular system instance.

**Definition 6.** A system scenario is a multiset of protocol roles.

A system scenario determines which sessions are present, and which agents play which roles.

*Example 7.* Consider the following simple system scenario, where *init* and *resp* are the roles defined in Example 5: This scenario is obtained by partially instantiating the above roles. The initiator is played by *a*, using fresh nonce *na* and shared key *k<sub>lt</sub>*, while the responder is *b*, using a fresh session key *k<sub>st</sub>*.  $\square$

We further require that each protocol role in a system scenario satisfies the *origination assumption*: all uninstantiated variables need to occur in a receive event before they occur in a send event or a status event (see [23] for details).

*Intruder's knowledge and valid traces* Let *S* be a system scenario. We say that *S'* is a *subscenario* of *S* if for every role *r' ∈ S'* there is a role *r ∈ S* s.t. *r'* is a prefix of *r*. We then say that a trace *tr* is *derived* from *S* if there exists an instance *S'* of *S* s.t. *tr* is an interleaving of a subscenario of *S'*. When the protocol is executed in presence of the intruder, we apply the Dolev-Yao model: (a) every message sent by an honest principal is added to the intruder's knowledge, and (b) every message received by an honest principal is produced by the intruder using the knowledge accumulated until that point. Formally, after the events in *tr* have taken place, the knowledge of the intruder is equal to  $IK \cup K(tr)$  where *IK* is the set of ground terms representing the initial intruder's knowledge, and  $K(tr) = \{m \mid \langle a : m \triangleright b \rangle \in tr\}$ . *IK* contains at least the intruder's identity *e*, and may contain other agent identities (*a, b, ...*) and public keys or nonces.

Suppose we have a ground trace  $tr = \langle tr' \ ev \rangle$ , with  $ev = \langle a : m \triangleleft b \rangle$ . By (b) above, we can say that the event *ev* in *tr* is *valid* if the intruder could produce *m* using  $IK \cup K(tr')$ . A whole trace is valid when all its receive communication events are valid, as shown in the next definition.

**Definition 8.** A ground trace *tr* is valid wrt *IK* if *tr* is empty, or for each  $i \in [0, \text{length}(tr) - 1]$ :

$$\text{last}(tr_{i+1}) = \langle a : m \triangleleft b \rangle \text{ implies that } m \in \mathcal{F}(K(tr_i) \cup IK)$$

This validity notion allows us to understand what a symbolic (i.e. non-ground) trace represents, namely all its *valid* instances. Given a (symbolic) trace *tr* and a set of ground terms *IK*, we let:

$$V(tr, IK) = \{tr' \mid tr' \text{ is a valid trace wrt } IK, \text{ and } tr' \text{ is an instance of } tr\}$$

### 3 Constraint Solving

To analyze a protocol, we execute a system scenario together with an intruder with the abilities of the DY rules, using a set of *constraints*. A constraint is a pair  $m : K$ , of a term *m* and a term set *K* (standing for *knowledge*). A constraint is called *simple* if *m* is a variable, ie.  $m \in \mathcal{V}$ . A *constraint set CS* is a finite set of constraints; *CS* is simple if each constraint in the set is simple.

**Definition 9.** Let  $CS = \{m_i : K_i\}$  be a constraint set, and let  $\sigma$  be a ground substitution for all the variables in *CS*. We say that  $\sigma$  is a *solution* of *CS* if for each *i*,  $m_i\sigma \in \mathcal{F}(K_i\sigma)$ . We also say that *CS* is *solvable* if there exists at least one solution  $\sigma$  of *CS*. A *partial solution*  $\gamma$  of *CS* is a substitution s.t.  $\text{dom}(\gamma) \subseteq \text{var}(CS)$ , and  $CS\gamma$  is *solvable*.

Recall that  $e \in IK$  (the intruder knows its own name initially). In the sequel, we consider only constraints  $m : K$  in which  $IK \subseteq K$ . This implies that a simple constraint is always solvable: the mapping  $\sigma(v_i) = e$  is always a solution of the simple constraint  $v_i : K_i$ . Millen and Shmatikov's reduction algorithm (called **P** in the following) [23] reduces a constraint set  $CS$  to (possibly empty set of) pairs of simple constraint sets  $CS'$  and substitutions  $\gamma$ . Millen and Shmatikov's result is:

**Theorem 10 ([23]).** (a) **P** always terminates. (b) Soundness: If **P** applied to  $CS$  outputs  $(CS', \gamma)$ , then  $\gamma$  is a partial solution of  $CS$ , and every solution of  $CS'$  is also a solution of  $CS\gamma$ . (c) Completeness: If  $CS$  is solvable with solution  $\sigma$ , then applying **P** to  $CS$  returns some  $(CS', \gamma)$  such that, for some solution  $\sigma'$  of  $CS'$ ,  $\sigma = \gamma\sigma'$ .

We now describe an algorithm which –given a system scenario  $S$  and initial intruder's knowledge  $IK$ – non-deterministically produces a set of symbolic traces, as described in [7]. This algorithm is based on [23], but instead of considering every *complete* interleaving of events, as described in [23], we incrementally add events during an execution, checking that the constraint set remains solvable. This procedure results in a significant efficiency gain wrt the original procedure. In the following we extend the original procedure by considering the new status events introduced in Definition 2.

**Procedure 11.** A state is a 4-tuple  $\langle S, IK, CS, tr \rangle$ , where  $S$  is a system scenario,  $IK$  is the initial intruder's knowledge,  $CS$  is a simple constraint set and  $tr$  is a (possibly non-ground) trace. A step from state  $\langle S, IK, CS, tr \rangle$  to  $\langle S', IK, CS', tr' \rangle$  is obtained by performing the following:

- Choose non-deterministically a non-empty role  $r \in S$ . Let  $r = \langle ev \ r' \rangle$ . Consider the following cases for  $ev$ :
  1. If  $ev$  is a status or a send communication event, let  $\gamma$  be the empty substitution and  $CS''$  be  $CS$ .
  2. If  $ev$  is a receive communication event, ie.  $ev = \langle a : m \triangleleft b \rangle$ , check if the intruder can generate  $m$  using the knowledge  $K(tr) \cup IK$ , by applying procedure **P** to  $CS \cup \{m : (K(tr) \cup IK)\}$ . If it is solvable, we obtain a new simple constraint set  $CS''$  and a partial solution  $\gamma$  (As there may be many  $CS''$  and  $\gamma$ , this step may result in branching).
- Let  $S' := (S \setminus \{r\} \cup \{r'\})\gamma$ ,  $CS' := CS''$  and  $tr' := \langle tr\gamma \ ev\gamma \rangle$ .

A run for  $S$  (with initial intruder's knowledge  $IK$ ) is a sequence of steps, starting from state  $\langle S, IK, \emptyset, \langle \rangle \rangle$ .

It is easy to see that this procedure terminates, since system scenarios are finite, and procedure **P** only outputs a finite number of partial solutions. Moreover, from Theorem 10 it follows that this procedure is correct and complete.

**Proposition 12.** For Procedure 11, it holds:

1. Soundness: Let  $\langle S', IK, CS, tr \rangle$  be a state in a run for  $S$  with initial intruder's knowledge  $IK$ . Then for every solution  $\sigma$  of  $CS$ ,  $tr\sigma$  is valid wrt  $IK$  and  $tr\sigma$  is derived from  $S$ .
2. Completeness: Let  $tr$  be a valid trace wrt  $IK$  derived from  $S$ . Then there exists a state  $\langle S', IK, CS, tr' \rangle$  in a run for  $S$  wrt  $IK$  and a substitution  $\sigma$  s.t.  $\sigma$  is a solution of  $CS$  and  $tr = tr'\sigma$ .

## 4 PS-LTL

We now introduce our language for writing security properties. Then we provide a semantics, in the form of *concrete* and *symbolic* validity.

**Definition 13.** A PS-LTL formula is defined by the following grammar:

$$\begin{aligned} \phi ::= & \text{true} \mid \text{false} \mid p(d_1, \dots, d_n) \mid \text{learn}(m) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \exists v.\phi \mid \\ & \forall v.\phi \mid Y\phi \mid \phi S\phi \end{aligned}$$

where each  $d_i$  ( $i \in [1, n]$ ) and  $m$  are either variables in  $\mathcal{V}$  or ground terms in  $\mathcal{T}^+$ .

Standard formulas **true**, **false**,  $\neg\phi$ ,  $\phi \wedge \phi$ ,  $\phi \vee \phi$  carry the usual meaning. Formula  $p(d_1, \dots, d_n)$  is a status event. **learn**( $m$ ) is a predicate stating that the intruder knows term  $m$  (we borrow the name from NPATRL [24]).  $Y\phi$  means ‘yesterday  $\phi$  held’, while  $\phi_1 S \phi_2$  means that ‘ $\phi_1$  held ever since a moment in which  $\phi_2$  held’. When  $v \in \mathcal{V}$ , we write  $\exists v.\phi$  and  $\forall v.\phi$  to bind  $v$  in  $\phi$ ;  $v$  represents terms. Other operators can be represented using the above defined operators:  $\phi_1 \rightarrow \phi_2$  is defined as  $\neg\phi_1 \vee \phi_2$ ;  $O\phi$  (*once*  $\phi$ ) is a shorthand for **true**  $S \phi$  and finally  $H\phi$  (*historically*  $\phi$ ) is a shorthand for  $\neg O\neg\phi$ . For clarity, we impose a precedence hierarchy for operators: unary operators bind stronger than binary operators. Operators  $Y$ ,  $O$ , and  $H$  bind equally strong and bind stronger than  $\neg$ . The precedence hierarchy for binary operators is  $S > \wedge > \vee > \rightarrow$ , where  $op_1 > op_2$  means “ $op_1$  binds stronger than  $op_2$ ”. In the sequel, we assume that PS-LTL formulas are *closed* (ie. they contain no free variables), and that each variable is quantified at most once (this can always be achieved using alpha conversion).

Our semantics  $\langle tr, IK \rangle \models \phi$  is defined for two different cases: First, we define it when  $tr$  is a ground trace, which we call *concrete* validity. Then, we extend it to the general case, in which  $tr$  may be symbolic. This establishes *symbolic* validity.

**Definition 14 (Concrete validity).**

Let  $\phi$  be a closed PS-LTL formula,  $tr$  be a ground trace and  $IK$  be an initial intruder’s knowledge. We then define  $\langle tr, IK \rangle \models \phi$  as:

$$\begin{aligned} \langle tr, IK \rangle \models \text{true} & \quad \text{and } \langle tr, IK \rangle \not\models \text{false} \\ \langle tr, IK \rangle \models p(d_1, \dots, d_n) & \text{ iff } tr = \langle tr' ev_2 \rangle \text{ and } p(d_1, \dots, d_n) = ev_2 \\ \langle tr, IK \rangle \models \text{learn}(m) & \text{ iff } m \in \mathcal{F}(K(tr) \cup IK) \\ \langle tr, IK \rangle \models \neg\phi & \text{ iff } \langle tr, IK \rangle \not\models \phi \\ \langle tr, IK \rangle \models \exists v.\phi & \text{ iff } \exists t \in \mathcal{T}^+ : \langle tr, IK \rangle \models \phi[t/v] \\ \langle tr, IK \rangle \models \forall v.\phi & \text{ iff } \forall t \in \mathcal{T}^+ : \langle tr, IK \rangle \models \phi[t/v] \\ \langle tr, IK \rangle \models \phi_1 \wedge \phi_2 & \text{ iff } \langle tr, IK \rangle \models \phi_1 \text{ and } \langle tr, IK \rangle \models \phi_2 \\ \langle tr, IK \rangle \models \phi_1 \vee \phi_2 & \text{ iff } \langle tr, IK \rangle \models \phi_1 \text{ or } \langle tr, IK \rangle \models \phi_2 \\ \langle tr, IK \rangle \models Y\phi & \text{ iff } tr = \langle tr' ev \rangle \text{ and } \langle tr', IK \rangle \models \phi \\ \langle tr, IK \rangle \models \phi_1 S \phi_2 & \text{ iff } \exists i \in [0, \text{length}(tr)] : (\langle tr_i, IK \rangle \models \phi_2 \wedge \\ & \quad \forall j \in [i+1, \text{length}(tr)] : \langle tr_j, IK \rangle \models \phi_1) \end{aligned}$$

Given our PS-LTL semantics, we can confirm our definitions for  $O$  and  $H$ .

**Proposition 15.** For every trace  $tr$ ,  $IK$  and closed PS-LTL formula  $\phi$ , (i)  $\langle tr, IK \rangle \models O\phi$  iff  $\exists i \in [0, \text{length}(tr)] : \langle tr_i, IK \rangle \models \phi$ , and (ii)  $\langle tr, IK \rangle \models H\phi$  iff  $\forall i \in [0, \text{length}(tr)] : \langle tr_i, IK \rangle \models \phi$ .

In fact, we can also state and prove other standard results for LTL and infinite traces, like the tautology  $H\phi \rightarrow O\phi$ . Furthermore, we can state some particular relations of PS-LTL :

**Proposition 16.** *For every trace  $tr$ ,  $IK$  and message  $m$ : (i)  $\langle tr, IK \rangle \models \text{learn}(m)$  iff  $\langle tr, IK \rangle \models \mathcal{O} \text{ learn}(m)$  iff  $\langle tr, IK \rangle \models \text{learn}(m) \mathcal{S} \text{ learn}(m)$ , and (ii)  $\langle tr, IK \rangle \models Y \text{ learn}(m)$  implies  $\langle tr, IK \rangle \models \text{learn}(m)$ .*

The proposition intuitively shows that the intruder never forgets information, and it follows from the monotonicity of  $\mathcal{F}(\cdot)$ , ie.  $\mathcal{F}(K(tr_i) \cup IK) \subseteq \mathcal{F}(K(tr_j) \cup IK)$  for each  $i \leq j$ .

Concrete validity is defined wrt ground traces. Therefore, to find a particular trace representing an attack, we would need to enumerate all possible ground traces derived from a scenario, which are infinitely many. To cope with this, in the sequel we introduce a method for checking validity wrt a *symbolic* trace, which represents (infinitely) many concrete traces.

**Definition 17 (Symbolic validity).** *Given a trace  $tr$  derived from a system scenario  $S$  and  $IK$ , we say that  $\langle tr, IK \rangle \models \phi$  iff  $\forall tr' \in V(tr, IK) : \langle tr', IK \rangle \models \phi$ .*

## 5 Checking Symbolic Validity in Constraint Solving

Let  $\varphi$  be a closed  $\mathcal{PS}$ -LTL formula representing a security property. We let  $A_\varphi = \neg\varphi$  be its corresponding *attack* property. Given a symbolic trace  $tr$  and  $IK$ , we define a procedure  $D$  that tries to find a ground instance  $tr'$  of  $tr$  s.t.  $\langle tr', IK \rangle \models A_\varphi$ . If  $D$  succeeds,  $tr'$  represents a violation of  $\varphi$  (hence an attack), since  $\langle tr', IK \rangle \models A_\varphi$  iff  $\langle tr', IK \rangle \not\models \varphi$ , and thus  $\langle tr, IK \rangle \not\models \varphi$ . On the other hand, if  $D$  fails, then we know that there is no  $tr'$  s.t.  $\langle tr', IK \rangle \models A_\varphi$ . In other words, for every ground instance  $tr'$  of  $tr$ ,  $\langle tr', IK \rangle \models \varphi$ , i.e.,  $\langle tr, IK \rangle \models \varphi$ . Thus  $D$  decides symbolic validity.

Our approach consists of two stages. We first translate a closed  $\mathcal{PS}$ -LTL formula  $\phi$  into an *elementary formula* EF, using a transformation  $T$ . Then, we input the translated formula to the decision procedure  $D$ .

**Definition 18.** *Elementary formulas EF (ranged over by  $\pi$ ) are defined by the grammar:*

$$\pi ::= \text{true} \mid \text{false} \mid t_1 = t_2 \mid m : K \mid \neg\pi \mid \pi \wedge \pi \mid \pi \vee \pi \mid \exists v.\pi \mid \forall v.\pi$$

Here  $t_1, t_2$  and  $m$  are either variables or ground terms,  $K$  is a set of terms and  $v$  is a variable.

Let  $\pi$  be an EF formula. Then we define its *left* free variables  $free_l(\pi)$  and its *right* free variables  $free_r(\pi)$ , as follows:

$$\begin{aligned} free_l(\text{true}) &= free_l(\text{false}) &&= \emptyset \\ free_l(t_1 = t_2) &&&= var(t_1) \\ free_l(m : K) &&&= var(m) \\ free_l(\neg\pi) &&&= free_l(\pi) \\ free_l(\pi_1 \wedge \pi_2) &= free_l(\pi_1 \vee \pi_2) &&= free_l(\pi_1) \cup free_l(\pi_2) \\ free_l(\exists v.\pi) &= free_l(\forall v.\pi) &&= free_l(\pi) \setminus \{v\} \end{aligned}$$

$free_r(\pi)$  is similar, but with:  $free_r(t_1 = t_2) = var(t_2)$  and  $free_r(m : K) = var(K)$ . We now give a semantics of an EF formula  $\pi$  wrt. a ground substitution  $\sigma$ .

**Definition 19.** *Let  $free_l(\pi) = \emptyset$  and  $free_r(\pi) = dom(\sigma)$ . Then  $\sigma \models' \pi$  is defined by:*



$$\begin{aligned}
\sigma \models' \text{true} & \quad \text{and } \sigma \not\models' \text{false} \\
\sigma \models' t_1 = t_2 & \quad \text{iff } t_1 = t_2\sigma \\
\sigma \models' m : K & \quad \text{iff } m \in \mathcal{F}(K\sigma) \\
\sigma \models' \pi_1 \wedge \pi_2 & \quad \text{iff } \sigma \models' \pi_1 \text{ and } \sigma \models' \pi_2 \\
\sigma \models' \pi_1 \vee \pi_2 & \quad \text{iff } \sigma \models' \pi_1 \text{ or } \sigma \models' \pi_2 \\
\sigma \models' \exists v. \pi & \quad \text{iff } \exists t \in \mathcal{T}^+ : \sigma \models' \pi[t/v] \\
\sigma \models' \forall v. \pi & \quad \text{iff } \forall t \in \mathcal{T}^+ : \sigma \models' \pi[t/v]
\end{aligned}$$

### 5.1 First Stage: $T(\phi, tr, IK)$

We define a translation  $T(\phi, tr, IK)$  from a  $\mathcal{PS}$ -LTL formula  $\phi$ , a trace  $tr$  and an initial intruder's knowledge  $IK$  into an EF formula:

**Definition 20.**  $T(\phi, tr, IK)$  is an EF formula resulting from applying the following three steps:

1. First, we repeatedly apply transformation  $[\cdot]$ , defined below, until none of the rules can be applied:

$$\begin{aligned}
[\exists v. \phi]tr & \Rightarrow \exists v. [\phi]tr \\
[\forall v. \phi]tr & \Rightarrow \forall v. [\phi]tr \\
[\neg \phi]tr & \Rightarrow \neg [\phi]tr \\
[\phi_1 \wedge \phi_2]tr & \Rightarrow [\phi_1]tr \wedge [\phi_2]tr \\
[\phi_1 \vee \phi_2]tr & \Rightarrow [\phi_1]tr \vee [\phi_2]tr \\
[Y\phi]\langle \rangle & \Rightarrow \text{false} \\
[Y\phi]\langle tr \ e \rangle & \Rightarrow [\phi]tr \\
[\phi_1 S \phi_2]\langle \rangle & \Rightarrow [\phi_2]\langle \rangle \\
[\phi_1 S \phi_2]\langle tr \ e \rangle & \Rightarrow [\phi_2]\langle tr \ e \rangle \vee ([\phi_1]\langle tr \ e \rangle \wedge [\phi_1 S \phi_2]tr) \\
[\text{true}]tr & \Rightarrow \text{true} \\
[\text{false}]tr & \Rightarrow \text{false} \\
[\text{learn}(m)]tr & \Rightarrow m : (K(tr) \cup IK) \\
[ev]\langle \rangle & \Rightarrow \text{false} \\
[p(D_1, \dots, D_n)]\langle tr \ q(E_1, \dots, E_m) \rangle & \Rightarrow \text{false if } p \neq q \text{ or } n \neq m \\
[p(D_1, \dots, D_n)]\langle tr \ p(E_1, \dots, E_n) \rangle & \Rightarrow D_1 = E_1 \wedge \dots \wedge D_n = E_n
\end{aligned}$$

2. Repeatedly rewrite atoms  $\neg \neg \phi$  to  $\phi$ , and move  $\neg$ 's inside conjunctions and disjunctions using DeMorgan distributive laws.
3. Move  $\forall$  quantifiers as far as possible to the right, and simplify universally quantified formulas over (possibly negated) equalities and constraints, according to the following rules:

$$\begin{aligned}
\forall v. (\phi_1 \wedge \phi_2) & \Rightarrow \forall v. \phi_1 \wedge \forall v. \phi_2 \\
\forall v. (\phi_1 \vee \phi_2) & \Rightarrow \forall v. \phi_1 \vee \forall v. \phi_2 \text{ if } v \text{ is not free in } \phi_1 \\
& \quad \text{or } v \text{ is not free in } \phi_2 \\
\forall v. \phi & \Rightarrow \phi \text{ if } v \text{ is not free in } \phi \\
\forall v. \neg(v = t) & \Rightarrow \text{false} \\
\forall v. (v = t) & \Rightarrow \text{false} \\
\forall v. (v : K) & \Rightarrow \text{false} \\
\forall v. \neg(v : K) & \Rightarrow \text{false}
\end{aligned}$$

One can check that translation  $T$  terminates, given a finite trace.

We call an EF formula *existential* if it is of the form  $\exists v_1 \dots \exists v_n. \varphi$ , and  $\varphi$  does not contain any quantifiers ( $\forall$  nor  $\exists$ ). We now define the subset  $\Phi$  of  $\mathcal{PS}$ -LTL over which we are going to decide symbolic validity:

$$\Phi \triangleq \{ \phi \mid \phi \text{ is a closed } \mathcal{PS}\text{-LTL formula and} \\ T(\phi, tr, IK) \text{ is existential for every trace } tr \text{ and } IK \}$$

We shall see that  $\Phi$  is still expressive enough for several interesting security properties. In particular, every property  $\varphi$  considered in Section 6 satisfies  $A_\varphi \in \Phi$ . Examples of  $\phi \notin \Phi$  are  $\phi_1 = \forall x. \exists y. (run(x) \wedge run(y))$ , for  $tr_1 = \langle \langle run(z) \rangle \rangle$  we have  $T(\phi_1, tr_1, IK) = \forall x. \exists y. (x = z) \wedge (y = x)$ , and  $\phi_2 = \exists x. \forall y. (run(x, y) \vee run(y, x))$ , for  $tr_2 = \langle \langle run(z, w) \rangle \rangle$  we have  $T(\phi_2, tr_2, IK) = \exists x. \forall y. ((x = z \wedge y = w) \vee (y = z \wedge x = w))$  for any  $IK$ .

The following lemma states that the translation  $T$  preserves the semantics of  $\mathcal{PS}$ -LTL wrt semantics of EF.

**Lemma 21.** *Let  $\phi$  be a closed  $\mathcal{PS}$ -LTL formula,  $tr$  be a trace and  $IK$  be an initial intruder's knowledge, and let  $\sigma$  be a substitution such that  $dom(\sigma) = var(tr)$ . Then  $\langle tr\sigma, IK \rangle \models \phi$  iff  $\sigma \models' T(\phi, tr, IK)$ .*

## 5.2 Second Stage: $D(\pi, CS)$

Given an existential EF formula  $\pi = \exists v_1 \dots \exists v_n. \varphi$ , we transform  $\varphi$  into its *disjunctive normal form*  $\varphi = \bigvee_j \psi_j$ , with  $\psi_j = \bigwedge_i \pi_{j,i}$ . Procedure  $D(\pi, CS)$  tries to find a disjunct  $\psi_j$  and  $\sigma$  that makes  $\psi_j$  (and therefore  $\varphi$ ) holds. For simplicity, we assume that each  $\psi_j$  contains just one *positive equality*  $L_{j,i} = R_{j,i}$ , one *negative equality*  $\neg(L_{j,i}^\neg = R_{j,i}^\neg)$ , one *positive constraint*  $m_{j,i} : K_{j,i}$  and one *negative constraint*  $\neg(m_{j,i}^\neg : K_{j,i}^\neg)$ . The generalization to the case with several atomic formulas and with (possibly negated) **true** and **false** atoms is straightforward.

**Procedure 22.** *Let  $CS$  be a simple constraint set. Let  $\psi_j = (L_{j,i} = R_{j,i}) \wedge \neg(L_{j,i}^\neg = R_{j,i}^\neg) \wedge (m_{j,i} : K_{j,i}) \wedge \neg(m_{j,i}^\neg : K_{j,i}^\neg)$ .*

1. Pick a disjunct  $\psi_j$  while possible, otherwise exit and return false.
2. Solve Positive Equality: Take a relevant most general unifier  $\rho$  of  $L_{j,i}$  and  $R_{j,i}$  such that  $dom(\rho) \subseteq var(L_{j,i}) \cup var(R_{j,i})$ , ie.  $L_{j,i}\rho = R_{j,i}\rho$  (If no mgu exists, go back to Step 1).
3. Solve Positive Constraint: Apply **P** to  $(CS \cup \{m_{j,i} : K_{j,i}\})\rho$ . Let  $\rho_1, \dots, \rho_l$  be the partial solutions.
4. Pick  $\rho_k$  while possible, otherwise go back to Step 1.
5. Solve Negative Constraint: Apply **P** to  $(CS \cup \{m_{j,i} : K_{j,i}, m_{j,i}^\neg : K_{j,i}^\neg\})\rho_k$ . If it is solvable, go back to Step 4.
6. Solve Negative Equality: Find a substitution  $\gamma$  s.t.  $L_{j,i}^\neg\rho_k\gamma$  and  $R_{j,i}^\neg\rho_k\gamma$  are ground and  $L_{j,i}^\neg\rho_k\gamma \neq R_{j,i}^\neg\rho_k\gamma$ , with  $(CS \cup \{m_{j,i} : K_{j,i}\})\rho_k\gamma$  solvable. If no  $\gamma$  is found, go back to Step 4.

Step 2 tries to solve the positive equality, finding a suitable unifier  $\rho$ . (We need a unifier and not a matching for the general case of many equalities). In case  $\rho$  is not found, then we can not make the disjunct to hold, so we try a different one going back to Step 1. Similarly, Step 3 solves the positive constraint. Step 5 checks that both  $\rho$  and  $\rho_k$  make the negative constraint to hold, checking that it is not solvable. Finally, Step 6 looks for a substitution that solves the negative equality. This step is the most difficult step to achieve, however checking syntactic equality is enough.

**Claim 23.**  *$L_{j,i}^\neg\rho\rho_k$  and  $R_{j,i}^\neg\rho\rho_k$  differ syntactically iff there exists  $\gamma$  s.t.  $L_{j,i}^\neg\rho\rho_k\gamma$  and  $R_{j,i}^\neg\rho\rho_k\gamma$  are ground and  $L_{j,i}^\neg\rho\rho_k\gamma \neq R_{j,i}^\neg\rho\rho_k\gamma$ , with  $(CS \cup \{m_{j,i} : K_{j,i}\})\rho\rho_k\gamma$  solvable.*

Since we consider (i) relevant unifiers for Step 2, which are only finitely many, (ii) **P** only outputs a finite number of solutions for Step 3 and 5, and (iii) by Claim 23 we only need to perform a syntactic check for Step 6, then we can deduce that procedure *D* terminates. The correctness of *D* is more challenging to establish.

**Lemma 24.** *Let  $\pi = T(\phi, tr, IK)$  be an existential elementary formula and  $\pi = \exists v_1 \dots \exists v_n. \bigvee_j \psi_j$ , with  $\psi_j = \bigwedge_i \pi_{j,i}$ . If  $D(\pi, CS)$  holds, then  $\exists \sigma : \sigma \models' \pi$ , with  $tr\sigma$  valid wrt *IK*.*

Now we are ready to state the main result of this section, which states that applying the transformation *T* of Definition 20 and *D* defined in Procedure 22 is sound.

**Theorem 25.** *Let *S* be a system scenario, *IK* be an initial intruder's knowledge,  $\phi$  a closed  $\mathcal{PS}$ -LTL formula representing a security property, and let  $A_\phi = \neg\phi$ . Let the system execute to a state  $\langle S', IK, CS, tr \rangle$ . Assume  $\pi = T(A_\phi, tr, IK)$  is existential,  $\pi = \exists v_1 \dots \exists v_n. \bigvee_j \psi_j$ , with  $\psi_j = \bigwedge_i \pi_{j,i}$ . Then, if  $\langle tr, IK \rangle \models \phi$ ,  $D(\pi, CS)$  fails.*

*Integrating  $\mathcal{PS}$ -LTL to Constraint Solving* We integrate the checking of a closed  $\mathcal{PS}$ -LTL formula  $\phi$  (representing a security property) into the constraint-based protocol analysis approach described in Procedure 11.

**Procedure 26.** *A state of Procedure 26 is the 5-tuple  $\langle S, IK, CS, tr, \phi \rangle$ . Procedure 26 is obtained by adding one more step at the end of Procedure 11:*

- *Compute  $\pi = T(\neg\phi, tr, IK)$ , and check that  $\pi$  is existential. Evaluate  $D(\pi, CS)$ . If  $D(\pi, CS)$  holds, then the run stops and we output the current (offending) trace *tr*. Otherwise, continue the run.*

The additional step essentially checks whether *tr* in the current execution state can be instantiated to provide a solution of  $\neg\phi$ , that is, to *falsify*  $\phi$ : by Theorem 25, we know that if  $D(T(\neg\phi, tr, IK), CS)$  holds, then  $\langle tr, IK \rangle \not\models \phi$ . In that case, the procedure terminates and outputs the trace *tr* that shows an attack. Otherwise, the procedure proceeds until an attack or no attack is found. Since *T* and *D* terminate, and Procedure 11 terminates, Procedure 26 also terminates.

*Implementation* We have fully integrated  $\mathcal{PS}$ -LTL into our previous verifier [7]. An online demo is available [8]. Essentially we extend the previous implementation by including the translation *T* of a  $\mathcal{PS}$ -LTL formula and the implementation of the decision procedure *D*.

## 6 Writing Security Properties with $\mathcal{PS}$ -LTL

In this section we show how to specify several security properties in  $\mathcal{PS}$ -LTL including secrecy, freshness and authentication. We have successfully used our tool to check these properties for some scenarios from our running example. The complete specification of protocols, scenarios and properties we have tested can be found in [9].

### 6.1 Authentication

First we specify in  $\mathcal{PS}$ -LTL various forms of authentication, as defined in [19, 10]. We consider the authentication of an initiator to a responder. The converse case is similar.

*Aliveness* The aliveness property is the weakest form of authentication in Lowe's hierarchy [19]:

*A protocol guarantees to a responder A aliveness of another agent B if, whenever A (acting as responder) completes a run of the protocol, apparently with initiator B, then B has previously been running the protocol.*

Notice that  $B$  may have run the protocol with an agent other than  $A$ . Moreover,  $B$  may not have been running the protocol recently. The aliveness of agent  $B$  to responder  $A$  can be specified in  $\mathcal{PS}$ -LTL as follows:

$$\forall A, B, D1, D2, D3. \exists A', R', D1', D2', D3'. \\ \text{end}(A, B, \text{responder}, D1, D2, D3) \rightarrow \mathbf{0} \text{ run}(B, A', R', D1', D2', D3')$$

We can check the aliveness property for our running example on a scenario  $S_1$  containing one initiator role and one responder role running in two different sessions:

$$S_1 = \{\text{init}(a, b, na, k_{lt}, K_{st}), \text{resp}(b, a, Nb, k_{lt}, k_{st})\}$$

Here, the initial intruder's knowledge is  $IK = \{b, e\}$ . We run scenario  $S_1$  and check the above formula with our tool and we obtained an attack, which corresponds to the one found by Lowe [17].

We also run our tool to check the aliveness property for the fixed version of BAN concrete Andrew Secure RPC protocol [17]. As claimed by Lowe [17], we also obtained no attack for this property using our tool with respect to scenario  $S_1$ .

*Weak Agreement* Weak agreement is defined as follows [19]:

*A protocol guarantees to a responder A weak agreement with another agent B if, whenever A (acting as responder) completes a run of the protocol, apparently with initiator B, then B has previously been running the protocol, **apparently with A**.*

For this property,  $B$  may not necessarily have been acting as initiator. The weak agreement property can be expressed in  $\mathcal{PS}$ -LTL as follows:

$$\forall A, B, D1, D2, D3. \exists R', D1', D2', D3'. \\ \text{end}(A, B, \text{responder}, D1, D2, D3) \rightarrow \mathbf{0} \text{ run}(B, A, R', D1', D2', D3')$$

Weak agreement is stronger than the aliveness property, therefore the attack mentioned above also applies to this property. The attack is also successfully reported by our tool.

*Non-injective Agreement* Non-injective agreement is defined as follows [19]:

*A protocol guarantees to a responder A non-injective agreement with another agent B on a set of data items D if, whenever A (acting as responder) completes a run of the protocol, apparently with initiator B, then B has previously been running the protocol, apparently with A, and B was acting as initiator in his run, and the two agents agreed on the data values corresponding to all the variables in D.*

This property is stronger than weak agreement. Nevertheless, it does not guarantee that there is a one-to-one relationship between the runs of  $A$  and the runs of  $B$ . The property we want to check can be formalized in  $\mathcal{PS}$ -LTL as follows:

$$\forall A, B, D1, D2, D3. \\ \text{end}(A, B, \text{responder}, D1, D2, D3) \rightarrow \mathbf{0} \text{ run}(B, A, \text{initiator}, D1, D2, D3)$$

As in the weak agreement, for the scenario  $S_1$  the aliveness attack also applies to this property and reported by our tool.

## 6.2 Secrecy

We now turn to study secrecy. In particular, we focus on two particular notions of secrecy, viz. standard secrecy and perfect forward secrecy.

*Standard secrecy* We define standard secrecy as the inability of an attacker to obtain the value of the secret [2]. Recall scenario  $S$  of our running example. The secrecy of the session key  $k_{st}$ , once the initiator  $a$  started a protocol run with the responder  $b$ , can be checked simply by the following  $\mathcal{PS}$ -LTL formula  $\neg learn(k_{st})$ . We checked this property using our tool and we found no secrecy attack on  $S$ .

*Perfect Forward Secrecy* We follow the definition of perfect forward secrecy (PFS) given by Diffie, et.al [13]:

*An (authenticated key exchange) protocol provides perfect forward secrecy if disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs.*

In Diffie et.al [13], the proposed Authenticated Diffie-Hellman key exchange protocol is shown to preserve PFS, since long term keys are only used to sign messages and are never related to the session key derivation. This is not the case for the RPC Andrew protocol and its variants, since the short term session key is directly encrypted by the long term key. Still, the specification of PFS in  $\mathcal{PS}$ -LTL is an interesting application of our logic because it exhibits the ability to use several **learn**'s in the same formula, c.f. related work.

In our framework, the disclosure of long-term secret keying material, e.g.  $k_{lt}$ , can be realized by providing an additional protocol role, which contains only one send event that leaks  $k_{lt}$  to the intruder. To specify PFS in  $\mathcal{PS}$ -LTL we need to ensure that (i) the leaking of the long-term key  $k_{lt}$  happens *after* a protocol run has been completed, and (ii) before the leaking of  $k_{lt}$  the short term session key ( $k_{st}$  in our scenario  $S$ ) has never been compromised. The former requirement allows us to have a completed run before the long-term key is disclosed. This enables us to check whether the key  $k_{st}$  exchanged in this completed run can be compromised by the disclosure of the long-term key  $k_{lt}$ . The latter serves as a precondition that guarantees the secrecy of the exchanged key before the disclosure of the long-term key. Let scenario  $S_2 = S \cup \langle \text{leaker} : k_{lt} \triangleright e \rangle$  and  $IK = \{e\}$ . We express PFS in  $\mathcal{PS}$ -LTL as follows:

$$learn(k_{lt}) \wedge Y(O(end(b, a, responder, n_a, k_{lt}, k_{st}) \wedge H \neg learn(k_{st}))) \rightarrow H \neg learn(k_{st})$$

Thanks to Proposition 16, we can rewrite the property in a more efficient form:

$$learn(k_{lt}) \wedge Y(O(end(b, a, responder, n_a, k_{lt}, k_{st}) \wedge \neg learn(k_{st}))) \rightarrow \neg learn(k_{st})$$

We run our tool to check this formula with respect to scenario  $S_2$  and we obtained an attack. In this attack, the disclosure of  $k_{lt}$  enables the attacker to compromise  $k_{st}$ .

## 6.3 Data Freshness

We state the data freshness property as follows:

*Data  $D$  is fresh whenever an agent  $A$  (either as an initiator or as a responder) never completes a protocol run with another agent agreeing on  $D$ , if once in the past  $A$  (either as an initiator or as a responder) has already completed a protocol run with*

another agent agreeing on the same data  $D$ .

The freshness of an exchanged session key  $K$  in our protocol is expressed in  $\mathcal{PS}$ -LTL as follows:

$$\forall A, B_1, R_1, N_1, K_1, K, B_2, R_2, N_2, K_2. \\ \mathbf{Y}(0 \text{ end}(A, B_1, R_1, N_1, K_1, K)) \rightarrow \neg \text{end}(A, B_2, R_2, N_2, K_2, K)$$

We run our tool to check the freshness of the session key  $k_{st}$  with respect to scenario  $S_1$ , and obtained an attack similar to the previous aliveness attack. In this attack, the session key  $k_{st}$  is used twice, i.e. when  $a$  was acting as an initiator in one session and as a responder in the other session. Thus, it violates the freshness of  $k_{st}$ .

For Lowe's fixed version of BAN concrete Andrew Secure RPC protocol [17], no attack was found with respect to the given scenario  $S_1$ .

## 7 Conclusions

We propose  $\mathcal{PS}$ -LTL, a language for specifying security properties.  $\mathcal{PS}$ -LTL is based on linear temporal logic (LTL) with pure-past operators, and it allows one to specify several security properties including authentication [18, 10] (*aliveness*, *weak agreement* and *non-injective agreement*), secrecy (*standard secrecy* [2] and *perfect forward secrecy* [13]) and also data freshness. We present a sound decision procedure to check a fragment of  $\mathcal{PS}$ -LTL against symbolic traces, thus allowing to attach a  $\mathcal{PS}$ -LTL interpreter into our protocol verification tool [7] thereby providing a full verification system. Due to space constraints we could not include the proofs in this version of the paper. They can be found in the full version [9].

*Future Work* There are many possible directions. One direction is to implement formula checking more efficiently: for example, such implementation would not recompute the translation of  $\mathcal{PS}$ -LTL to elementary formula EF every time a property is checked, but maintain an internal data structure which can be optimized as the trace gets expanded, in the lines of [16]. Another possible direction would be to enlarge the subclass  $\Phi$  of  $\mathcal{PS}$ -LTL thus obtaining a more expressive language (eg. to cover stronger authentication notions like the ones in [10]). Also, it is interesting to model more protocols and their properties, e.g. from the Clark and Jacob library [5] (we already tested four variants of the Andrew RPC protocol, and also the Needham-Schroeder Public Key protocol, see [8]). Finally, more in-depth comparison to other existing logics would be beneficial, such as NPATRL [24].

## References

1. D. Basin, S. Mödersheim, and L. Viganò. Constraint differentiation: A new reduction technique for constraint-based analysis of security protocols. In *CCS'03*, pages 335–344. ACM Press, New York, 2003.
2. B. Blanchet. Automatic proof of strong secrecy for security protocols. Research Report MPI-I-2004-NWG1-001, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, July 2004.
3. M. Bozzano, G. Delzanno, and M. Martelli. A bottom-up semantics for linear logic programs. In M. Gabbriellini and F. Pfenning, editors, *Proc. Second International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 92–102. ACM Press, 2000.

4. M. Burrows, M. Abadi, , and R.M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
5. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. <http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz>, 1997.
6. R. Corin, A. Durante, S. Etalle, and P. H. Hartel. A trace logic for local security properties. In *Int. Workshop on Software Verification and Validation (SVV)*, volume 118, pages 129–143, Mumbai, India, Dec 2003. Elsevier Science in Electronic Notes in Theoretical Computer Science.
7. R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th Int. Static Analysis Symp. (SAS)*, volume LNCS 2477, pages 326–341, Madrid, Spain, Sep 2002. Springer-Verlag, Berlin.
8. R. Corin, S. Etalle, and A. Saptawijaya. Online demo for  $\mathcal{PS}$ -LTL . At <http://130.89.144.15/cgi-bin/psltl/show.cgi>, June 2005.
9. R. Corin, S. Etalle, and A. Saptawijaya.  $\mathcal{PS}$ -LTL for constraint-based security protocol analysis. Long version of this paper, including Prolog code, at <http://www.cs.utwente.nl/corin/ces05long.ps>, June 2005.
10. C.J.F. Cremers, S. Mauw, and E.P. de Vink. Defining authentication in a trace model. In T. Dimitrakos and F. Martinelli, editors, *Fast 2003*, Proceedings of the first international Workshop on Formal Aspects in Security and Trust, pages 131–145, Pisa, September 2003. IITT-CNR technical report.
11. S. Delaune and F. Jacquemard. A decision procedure for the verification of security protocols with explicit destructors. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 278 – 287, 2004.
12. G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *11th Int. Logic Based Program Synthesis and Transformation (LOPSTR)*, volume LNCS 2372, pages 76–90, Paphos, Greece, Nov 2001. Springer-Verlag, Berlin.
13. W. Diffie, P. C. Van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107 – 125, June 1992.
14. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
15. A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4):435–484, 2004.
16. K. Havelund and G. Rosu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS, 2001.
17. G. Lowe. Some new attacks upon security protocols. In *Proceedings of the Computer Security Foundations Workshop VIII*, page 162, 1996.
18. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.
19. G. Lowe. A hierarchy of authentication specifications. *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 31, 1997.
20. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
21. C. Meadows, P. F. Syverson, , and I. Cervesato. Formal specification and analysis of the group domain of interpretation protocol using NPATRL and the NRL protocol analyzer. *Journal of Computer Security*, 12(6):893–931, 2004.
22. J. Millen. Constraint solving in Prolog Webpage. <http://www.csl.sri.com/users/millen/capsl/constraints.html>.
23. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, November 2001.
24. P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes and Cryptography*, 7:27 – 59, 1996.
25. F. J. Thayer, J. Herzog, , and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

## Masquerade Detection Using IA Network

Subrat Kumar Dash<sup>1</sup>, Sanjay Rawat<sup>\*3</sup>, G. Vijaya Kumari<sup>2</sup>, and Arun K. Pujari<sup>1</sup>

<sup>1</sup> AI Lab, Dept. of Computer & Information Sciences  
University of Hyderabad, Hyderabad- 500046 India  
`subrat.dash@gmail.com`; `akpcs@uohyd.ernet.in`,

<sup>2</sup> Dept. of Computer Science, JNTU, Hyderabad - 500072 India  
`vij.tej@hotmail.com`

<sup>3</sup> Intoto Software (I) Pvt. Ltd.  
Uma plaza, Nagarjuna Hills, Punjagutta  
Hyderabad-500082 India  
`sanjayr@intoto.com`

**Abstract.** In this paper we propose a novel masquerade detection method based on constraint satisfaction problem. A masquerade attack is a challenge to the computer security, where an illegitimate entity poses as (and assumes the identity of) a legitimate entity. The illegitimate user, called masquerader, hides his/her identity by impersonating a legitimate user in a computer system or network and may maliciously damage the system. The detection of a masquerader relies on a user signature, a sequence of commands collected from a legitimate user. The underlying assumption is that the signature captures detectable patterns in a user's sequence of commands. We model a user as a binary constraint network such that each node represents an episode of commands and binary relationship between a pair of episodes is encoded as the disjunction of the Allens Interval relations. The well-known framework IA network is employed for the detection purpose. Any new subsequence of commands should be consistent with at least one user network. If the subnetwork is not consistent with any of the known networks, then we identify the subsequence as masquerade. We make use of a novel technique of episode determination for this purpose. We performed extensive experimentation on a well-known dataset (Schonlau Dataset) and find encouraging results.

**Keywords:** Masquerader, Unix commands, Frequent episodes, Interval algebra, Constraint satisfaction problem.

## 1 Introduction

Advancement in the technology and computing is leading to better and more efficient solutions to problems. Due to high performing computing devices, more and more data can be analyzed rapidly for better understanding. But this high performing characteristics also has a dark side. It has become relatively easier

---

\* During the work, the author was associated with UoH as PhD scholar.



to capture the encrypted data and decrypt it very fast, which causes the disclosure of important information to unintended person. This has given rise to the problem of managing important information, like passwords, properly. Also, as a human being, we tend to choose mnemonic passwords, so that we can recall them easily. Therefore, it is always possible that the sensitive information, like password, be known to others and if so, the consequences are very much obvious to us.

One specific consequence of this information leakage is known as masquerade attack, where an illegitimate entity poses as (and assumes the identity of) a legitimate entity. The illegitimate user, called masquerader, hides his/her identity by impersonating a legitimate user in a computer system or network and may maliciously damage the system. Masquerade attack can occur in varieties of ways such as by obtaining a legitimate user's password, accessing an unattended and unlocked workstation, forging email address in messages, overtaking a computer via a network access. It is not possible to detect such attacks by any type of detection at the time of accessing. It is also hard to detect this type of security breach at its initiation because the attacker appears to be a normal user with valid authority and privileges. Masquerader can be either an insider with malicious intent trying to hide his identity by impersonating other users or an outsider, who generally try to gain access to the account of the super-user. The broad range of damage that can be caused via masquerade attacks makes this as one of most serious threats to computer and network infrastructure.

The detection of a masquerader relies on a user signature, a sequence of commands collected from a legitimate user. The underlying assumption is that the signature captures detectable patterns in a user's sequence of commands. This signature is compared to the current user's session. A sequence of commands produced by the legitimate user should match well with patterns in the user's signature, whereas a sequence of commands of a masquerader should match poorly with the user's signature. The detection becomes difficult when the masquerader perfectly mimics original user's behavior. There is also a chance that the legitimate user may be detected as a masquerader if the user's behavior change, which may cause annoying false alarms.

In the present paper, we propose a novel way of modeling user behavior and the detection of masqueraders. Each user is profiled in terms of the unix commands, issued by him. From the command history, the frequent episodes of the commands are calculated by using an algorithm, originally proposed in [3]. We make use of 13 temporal relations to find the relationships among various episodes in the command data[1]. These relationships are depicted as binary constraint network and each user is represented as one network. When a new sequence of commands is encountered, the corresponding constraint network is generated based on the episodes present in the sequence and the binary relationships among the episodes. The new network in conjunction with the user network is subjected to well known consistency checking technique of Temporal CSP. If the augmented network is consistent by itself but not consistent in conjunction with any of the user network then the sequence is identified as a

masquerade sequence. We employ novel approach of episode determination and temporal CSP techniques for this purpose.

The proposed methodology is tested on the well known Schonlau dataset [14]. The experimental results show the high accuracy of the proposed method.

The rest of the paper is organized as follows. In section 2 we briefly outline the existing techniques of masquerade detection. Section 3 gives a preliminary background about the episode discovery and interval algebra. We discuss about the proposed method in section 4. Section 5 is concerned with the experimental details. Our conclusion and future work follows in section 6.

## 2 Related Work

The detection of a masquerader relies on a user signature, a sequence of commands collected from a legitimate user. The underlying hypothesis is that a sequence of commands produced by the legitimate user should match well with patterns in the user signature, whereas a sequence of commands of a masquerader should match poorly with the user's signature. Based on this assumption, there have been numerous attempts at successfully detecting masquerade attacks (minimizing false negatives) without degrading the quality of a user's session (minimizing false positives).

Schonlau *et al.* [14] investigate the use of various techniques, like Bayes 1-Step Markov, Hybrid Multi-Step Markov, Incremental Probabilistic Action Modeling (IPAM), Uniqueness, Sequence-Match, and Compression for masquerade detection. Bayes 1-Step Markov method is based on single-step transitions from one command to the next, and it determines the consistency of the observed transition probabilities with historical probabilities. As reported in [14], this technique is the best performer in terms of correct detections, but failed to get close to the desired false alarm rate. Hybrid Multi-Step Markov method is based on Markov models. In some sense it is hybrid model of Markov model and a simple independence model, depending on the proportion of commands in the test data that are not observed in the training data. IPAM (incremental probabilistic action modeling) is based on single-step command transition probabilities, estimated from the training data. IPAM's performance reportedly ranks with those in the lowest-performing group. Uniqueness approach is based on the frequency of various commands. Commands that are not seen in the training data, may indicate a masquerade attempt. The more infrequently a command is used by the user community as a whole, the more indicative that command is of being used by a masquerader. It is reported that Uniqueness is a relatively poor performer in terms of detecting masqueraders, but is the only method able to approach the target false alarm rate of 1%. Sequence-Match method computes a similarity match between the most recent user commands and a user profile. On the Schonlau data, it is a poor performer. The idea behind the compression approach is that new data from a given user compresses at about the same ratio as old data from that same user, and that data from a masquerading user will com-

press at a different ratio and thereby be distinguished from the legitimate user. Compression was the worst performer of the methods tested.

Maxion and Townsend [11] propose some improvements over the methods proposed in [14]. Their Bayesian model assumes that the user generates a sequence of commands, one command at a time, each with a fixed probability that is independent of the commands preceding it. The probability for each command  $c$  for a given user  $u$  is based on the frequency with which that command appears in the training data. The probability of a sequence of command is the product of probabilities of individual commands in the sequence. A block is detected as legitimate or masquerade based on the ratio of its probability as the user  $u$  (*self*) or not the user  $u$  (*non-self*). In spite of the unrealistic assumption of independence of individual commands, the technique performs very well. As an enhancement, it is shown in [12] that valuable information is lost when truncated command line data is used. It is proposed to use enriched command data. The enriched commands include information like name, arguments, flag, alias, options, directory, and history. It is reported that the dataset with enriched commands yields better results than the earlier dataset of truncated commands.

In [4] Coull *et al.* propose a novel technique based on pair-wise sequence alignment, which is a variation of the classic Smith-Waterman algorithm for biological sequence [16]. It is observed that none of the conventional alignments like local alignment or, global alignment is suitable in their original form for the matching of command sequence. Therefore, in order to suit the context, a novel scoring system is proposed that rewards the alignment of commands in the user segment but does not necessarily penalize the misalignment of large portions of the signature. This method produces a hit rate of 75.8% and false positive rate of 7.7% that are extremely competitive with other top masquerade detection algorithms. The only algorithms that perform comparably with these results are the Naive Bayes algorithms.

Very recently, a new and efficient masquerade detection technique based on SVM is proposed by Kim and Cha [8]. It is based on two novel concepts of *common commands* and *voting engine*. The common commands are sets of commands used frequently by more than  $n$  number of users at the rate exceeding  $Y\%$ . In order to extract features, the blocks of 100 commands are further viewed as smaller blocks by sliding a smaller window within the block. Blocks of 100 commands were divided into six different sub-blocks, each containing 50 commands with a sliding window of size 10. SVM predictor determines if each sub-block is normal or not. A *voting engine* decides if the total block is to be considered as being anomalous. If the number of masquerade sub-blocks exceeds threshold value, the block is considered as masquerade block. The results are reported to be the best so far with 80.1% of detection rate and 9.7% false positive.

### 3 Preliminary Background

In this section, we provide necessary background to understand the proposed technique.

### 3.1 Frequent Episode Discovery

In this section we describe an algorithm to extract meaningful subsequences (episodes) from a continuous sequence of commands. An episode is defined as an ordered set of elements (commands) within a given interval, such that the order is maintained in whole data. The idea has been taken from the *Voting-Experts* paradigm, proposed in [3]. The episode discovery method is concerned with assigning score to every element of the sequence so that higher value of the score indicates more likelihood of the element being the end point of an episode. The scores for each element are accumulated for each position of a sliding window of fixed length. While the window slides from left to right, the boundary-expert scores for a position by computing boundary entropy and the frequency-expert votes for the position based on the frequency of occurrence of the subsequence in the whole sequence. The main intuition behind the boundary entropy is the following.

In a subsequence, if any element precedes many distinct elements then it is difficult to determine any pattern of occurrence of the pair of elements. Hence, the entropy at this element has a very high value. On the other hand, if there is any specific pattern of occurrence then the entropy would be low. Similarly, the frequency-expert assigns high score when the subsequence is very frequent, which is attributed to being more meaningful. In order to complete these scores efficiently, it is proposed to compile the sequence data in the form of a *trie* of *ngrams*. This data structure is used to determine the scores at every location. We describe below the construction of trie from the sequence data.

**Construction of Trie** The trie can be viewed as a *pre-fix tree* of depth  $d$ , so that each distinct subsequence of length  $d - 1$  is a path from root node to a leaf node in the tree. Two subsequences having common prefix share common ancestors representing the prefix fragment. At every node, the frequency indicates the frequency of the subsequence represented by the path from root to the current node. The algorithm for construction of the trie is given in figure 1.

We illustrate the concept with the following example.

Example 1: Let us consider the sequence of six commands: `<xrdb, cpp, sh, cpp, sh, mv>`. The trie with depth 3 can be generated using the algorithm as depicted in figure 2.

We can observe that the leaf node labeled `sh` (second from left) represents the sequence `{cpp, sh}` and hence the number 2 at this node indicates the frequency of the subsequence. And each of the sequences `{xrdb, cpp}`, `{sh, cpp}` and `{sh, mv}` is present once. The two sequences `{sh, cpp}` and `{sh, mv}`, have a common prefix `{sh}`, which is also the common ancestor for the corresponding nodes.

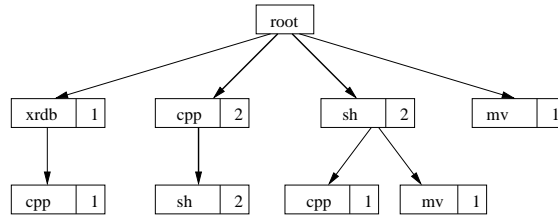
**Calculation of boundary entropy using the trie** The entropy of a node refers to the entropy of the sequence from the root node to the concerned node. Let  $f(x)$  be the frequency of the node  $x$ . Let  $x_0$  be a node and  $parent(x_0)$  be the

---

**Input:** Sequence of commands  $C$ , depth  $d$   
**Initialize:** root = NULL  
 $n = d - 1$   
**do** for each  $c_i \in C$   
    **if** root has a child node labeled  $c_i$  then  
        increment frequency of node  $c_i$  by 1  
    **else**  
        add new child node labeled  $c_i$  with frequency 1  
    **endif**  
**do** for  $j = i - n + 1$  to  $i - 1$   
    **if**  $j > 0$  then  
        **do** for each subsequence  $s_k$  comprising of commands  $c_j$  to  $c_{i-1}$   
            **if**  $s_k$  has a child node with labeled  $c_i$  then  
                increase frequency of this node by 1  
            **else**  
                add a new child node to the subsequence  $s_k$  labeled  $c_i$  with frequency 1  
            **endif**  
        **enddo**  
    **endif**  
**enddo**  
**enddo**

---

**Fig. 1.** Algorithm to construct an ngram of depth  $(n + 1)$  from a command sequence  $C$ .



**Fig. 2.** Trie for Example 1 with  $d = 3$ . The thickness of edges indicates the frequency.

parent node of  $x_0$ . Let us assume that  $x_1, x_2, \dots, x_m$  are the other child nodes of  $parent(x_0)$ .

The probability of the subsequence represented at node  $x_0$ , denoted as  $p(x_0)$ , is given by

$$p(x_0) = \frac{f(x_0)}{f(parent(x_0))} \quad (1)$$

The entropy of  $parent(x_0)$  is given by

$$e(parent(x_0)) = - \sum_{i=0}^m p(x_i) \log p(x_i) \quad (2)$$

It can be noted that the entropy for the leaf nodes is zero.

Each node of the  $n$ -gram trie has two parameters, one is frequency and the other is entropy (except the root node). Level 1 onwards, for each level, we calculate the mean frequency ( $f_l$ ), mean entropy ( $e_l$ ), standard deviation taking  $f_l$  ( $\sigma_{fl}$ ), and standard deviation taking  $e_l$  ( $\sigma_{el}$ ). These are calculated by taking the parameters of each node belonging to the same level. Now, for each node belonging to the same level we standardize its frequency ( $f$ ) and entropy ( $e$ ) as,

$$f = \frac{f - f_l}{\sigma_{fl}}, \text{ and } e = \frac{e - e_l}{\sigma_{el}} \quad (3)$$

**Finding episodes using the  $n$ -gram trie structure** To find episodes from the given command stream, it is necessary to find the correct boundary in the stream. We achieve this, by using the above  $n$ -gram trie with two parameters: frequency and entropy. Both the parameters contribute equally in finding the possible boundary by assigning scores to the probable boundary positions.

The above trie data structure helps us in efficiently computing the entropy and frequency of a subsequence. We take a window of size  $n$  ( $n + 1$  is the size of the trie) and examine different subsequences within the window. For instance, if  $x_0, x_1, x_2, \dots$ , and  $x_{n-1}$  are the elements falling in the window, then we examine the entropy at each location as follows.

The entropy at location  $i$  is the entropy of the node  $x_i$  at level  $i + 1$  along path  $x_0, x_1, \dots, x_i$ . The location corresponding to highest entropy is identified and its score is incremented by 1.

The frequency at location  $i$  is calculated by the sum of the frequencies of subsequences  $(x_0 \dots x_{i-1})$  and  $(x_i \dots x_{n-1})$ . The score at the location with highest frequency is incremented by 1. In this case, our goal is to maximize the sum of the frequencies of the left and right subsequences of the probable boundary.

We take a sliding window of length  $n$ . There are  $n$  possible boundary positions inside the window. After sliding the window across the whole command sequence, we end up with scores for each location in the sequence. In a stream of  $|C|$  commands, there are  $|C| - 1$  positions within the sequence. If a position is repeatedly voted for boundary by different windows then it is likely to accrue a locally-maximum score. We choose the position with local maximum of score as boundary of the episode.

### 3.2 Interval Algebra

Allen in his landmark paper [1] has proposed Interval Algebra, with 13 basic relations to relate any pair of time intervals in which events could occur. This initiated a substantial research activity in AI front to devise practical systems, which reason about time. The set of all basic relations in IA, is represented by,  $I = \{b, eq, m, o, d, s, f, bi, mi, oi, di, si, fi\}$ . These relations are exhaustive and are pair wise disjoint. Figure 3, gives the semantics of these basic relations. When the relation between a pair of intervals is indefinite, it is expressed as disjunction

of basic relations and is represented as a set. For example the relation  $\{m, o, s\}$  between events A and B represents the disjunction

$$(A \text{ meets } B) \vee (A \text{ overlaps } B) \vee (A \text{ starts } B).$$

Thus there are  $2^{13} = 8192$  possible ways to relate a pair of intervals. An IA network is a graphical representation of this information where the vertices represent events and directed edges are labeled with sets of basic relations. The main reasoning tasks in this framework include, checking consistency of the given information and finding the feasible relations among all the variables in the network. The temporal information represented in terms of a collection of qualitative relations constrains time intervals and the reasoning tasks therefore reduce to the standard Constraint Satisfaction Problem (CSP). A CSP consists of a set of constraints over a set of variables, where each variable is associated with its domain of values.

An IA network is a network of binary constraints where the variables represent time intervals, the domain of the variables are the end points of the variables and the binary constraints between variables are represented implicitly by the sets of basic relations.

Determining the feasible relations for example can be viewed as determining the deductive consequences of the given temporal information. For example from the information, **episode1 meets episode2**, **episode2 meets episode3**, we could derive that **episode1 before episode3**.

The main inference technique (path consistency) in this framework is based on constraint propagation.


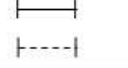
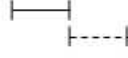
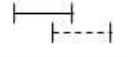
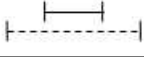
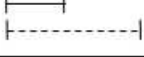
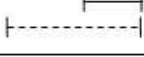
*Consider 3 intervals  $I, J, L$  with constraints  $I R_{ij} J, J R_{jl} L$  and  $I R_{il} L$ . Compute relational composition and intersect with the direct relation.*

$$I R_{il} L = (I R_{ij} J \otimes J R_{jl} L) \cap I R_{il} L.$$

Continue until fixed relation. This path consistency algorithm is used as inference algorithm for Allen's Interval Algebra.

## 4 Proposed Method

The present method of masquerade detection is based on the user command data. We observe that while a user shows a consistent behavior over a long period of time, it may happen that due to some requirement of temporary nature, the same user may type in few different commands. In such situation, we get interleaving of command sequence i.e. during user's usual command sequence, there may be some other command sequence, arisen due to temporary requirement. Under such conditions, mere command sequence matching may not be very suitable technique to apply. We, therefore, propose to use interval algebra to capture interleaving of different command subsequences. We consider user command data as time series and apply various temporal relations, depicted in figure 3, to find the relation among various command subsequence (we call as episodes).

Relation	Symbol	Inverse	Example
X before Y	b	bi	
X equal Y	=	=	
X meets Y	m	mi	
X overlaps Y	o	oi	
X during Y	d	di	
X starts Y	s	si	
X finishes Y	f	fi	

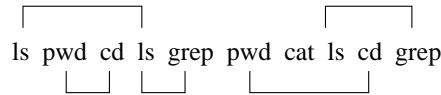
**Fig. 3.** 13 basic relations in IA

Let there be a total of  $K$  users. Once the command sequence for each user is collected, we apply the frequent episode discovery algorithm, described in figure 1, to find the frequent episodes of user command sequence for all user. Let there be a total of  $N$  episodes. These  $N$  episodes are represented as nodes of a directed graph  $G_i$  corresponding to user  $i$ . For each user and for each episode, we find the interleaving of episodes in user's command sequences by using the 13 relations shown in figure 3. The sets of relations among episodes that are being satisfied by the user's command sequence constitute the edges of the graph  $G_i$ . Thus for each user  $i$ , we have a graph  $G_i$  to represent the user's normal behavior. We illustrate the above method by taking an example below.

Let the user's command sequence be `<ls pwd cd ls grep pwd cat ls cd grep>`. Let us take the following three frequent episodes.

`(pwd cd)`, `(ls ls)` and `(ls grep)`

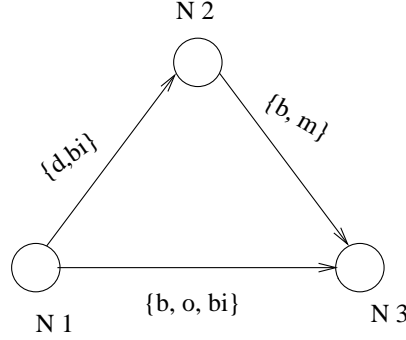
The interleaving of episodes is shown in figure 4



**Fig. 4.** The interleaving of the episodes in user command sequence.



On the basis of interleaving, shown in the figure 4, we get the following graph (figure 5)



**Fig. 5.** User's profile shown as the graph, where each node corresponds to one episode and each edge denoted the set of constraints, satisfied by corresponding nodes.

For the detection of masqueraders, the incoming command sequence, corresponding to user  $i$ , is also subjected to same procedure of forming the graph,  $G'_i$ , as mentioned above. The so formed new graph  $G'_i$  is compared with the user's graph  $G_i$  to find the consistency with the normal graph  $G_i$ . To do so, we compare the set of relations for each edge of the two graphs. The following expression is used for comparison.

$$(13 - edge(G_i)) \cap edge(G'_i) = \text{NULL} \quad (4)$$

If equation 4 holds, then incoming command sequence belongs to user  $i$ . The intuition behind the above equation is that, if the incoming command sequence indeed is coming from the genuine user, then it should also form the same tree and in such case the expression  $13 - edge(G_i)$  consists of relations not belonging to genuine user, whose intersection with incoming sequence, thus, gives information about the normal or masquerader. If the above relation (equation 4) does not hold then we go for *path consistency* check by using the Qualitative-Path-Consistency algorithm [6]. If the graph  $G'_i$  is consistent with the graph  $G_i$ , the incoming command sequence belongs to user  $i$  and if the graph  $G'_i$  is inconsistent with the graph  $G_i$ , the incoming command sequence does not belong to user  $i$ . But, if the graph corresponding to the new sequence data is NULL, we directly flag it as masquerade sequence without comparing with the normal graph.

In the next section, we report experimental results on Schonlau dataset.

## 5 Experimental Results

For experimentation, we choose Schonlau dataset [14], which is a *truncated* command dataset (i.e. excluding the arguments of commands), commonly called as

SEA dataset. The user's commands are collected by UNIX *acct auditing mechanism* consisting of 15,000 truncated commands for each of the 70 users. Out of these 70 users, 50 users are selected randomly. Commands entered by the rest 20 users are used to simulate masquerade activities. Each command set is decomposed into 150 blocks consisting of 100 commands each, and the first 50 blocks, or 5,000 commands, are used as training data and the rest as test data. Experiment administrators randomly inserted 0~24 command blocks as a means of approximating actions by masqueraders. The testing data is contaminated block-wise, so that a testing block is either contaminated completely or not at all.

From training data of all users, we find 9770 episodes. Out of these 9770, we select the episodes that occur at least 1000 times in the training data. From these episodes, we discard those episodes, which are the multiples of some smaller episode. After this preprocessing, we get 20 episodes for constructing user's profile  $G_i$ . We perform masquerade detection on testing data by taking one block of 100 commands at a time. To measure the accuracy, we define the following measures of accuracy.

$$\begin{aligned}\mu_i &= \frac{\# \text{ of normal block detected as normal}}{\text{total } \# \text{ of normal blocks}} \\ \lambda_i &= \frac{\# \text{ of masquerade block detected as masquerade}}{\text{total } \# \text{ of masquerade blocks}}\end{aligned}\tag{5}$$

The above expression is calculated for each user  $i$ . Once it is calculated, the total accuracy of the method is calculated as follows.

$$TotalAccuracy = \frac{\sum_{i=1}^K (\mu_i + \lambda_i)}{2K}\tag{6}$$

It should be noted that the total accuracy given by the equation 6 incorporates *true negatives* ( $\mu_i$ ) and *true positives* ( $\lambda_i$ ) in the parlance of intrusion detection. We also observe that though the total number of profiled user is  $K = 50$ , some of them do not contain any masquerade blocks. Such users are excluded while calculating the value of total accuracy. Based on the accuracy measure, defined by equation 6, we get an accuracy of 0.76 on the test data.

## 6 Conclusions

In the present study, we investigate the applicability of IA in the problem of masquerade detection. We observe that user's commands data has some variability from time to time for a short period of time. To capture the interleaving of various command sequences, we make use of 13 temporal relations and represent the user's profile as a graph. Each new command sequence is converted into a similar graph and is compared with the corresponding user's graph. If the graph is not consistent with the normal graph, we flag the new command

sequence as masquerade. The work is still in its preliminary stage and needs a lot of analysis and experimentation. We also observe that the variation in user's command sequence should also be considered while comparing it with new command sequence. We are trying to incorporate such things into our work, which form our future work.

### Acknowledgement

This research is supported by Ministry of Communication and IT, Govt of India under the grant no. 12(22)/04-IRSD dated: 04.02.2004.

### References

1. Allen, J.: Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26, (1983) 832-843
2. Chinchani, R., Muthukrishnan, A., Chandrasekaran, M., Upadhyaya, S.: RACOON: Rapidly generating user command data for anomaly detection from customizable templates. 20th Annual Computer Security Applications Conference (ACSAC), Tucson, AZ, December (2004)
3. Cohen, P., Heeringa, B., Adams, N. M.: An unsupervised algorithm for segmenting categorical timeseries into episodes. In: *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery*, London, UK. September (2002) 49-62
4. Coull, S., Branch, J., Szymanski, B., Breimer, E.: Intrusion detection: A bioinformatics approach. In: 19th Annual Computer Security Applications Conference, Las Vegas, Nevada, December 8-12. (2003)
5. Davison, B. D., Hirsh, H.: Predicting sequences of user actions. *Predicting the Future: AI Approaches to Time-Series Problems*. AAAI Technical Report WS-98-07, AAAI Press, Menlo Park, California, (1998)
6. Dechter, R.: *Constraint Processing*. Morgan Kaufmann Publishers. (2003)
7. Killhourhy, K. S., Maxion, R. A.: Investigating a possible flaw in a masquerade detection system. Technical Report CS-TR: 869, School of Computing Science, University of Newcastle. (2004)
8. Kim, H.-S., Cha, S.-D.: Empirical evaluation of SVM-based masquerade detection using UNIX commands. *Computers & Security*, Vol. 24, March. (2005) 160-168
9. Lane, T., Brodley, C. E.: Temporal Sequence Learning and Data Reduction for Anomaly Detection. In: *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, San Francisco, California, November 3-5. (1998) 150-158
10. Maxion, R. A., Townsend, T. N.: Masquerade detection augmented with error analysis. *IEEE Transactions on Reliability*, 53(1) March (2004) 124-147
11. Maxion, R. A., Townsend, T. N.: Masquerade detection using truncated command lines. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN-02)*, Washington, D.C. 23-26 June (2002) 219-228
12. Maxion, R. A.: Masquerade detection using enriched command lines. In: *International Conference on Dependable Systems and Networks (DSN-03)*, San Francisco, CA, USA, June (2003)

13. McCallum, A., Nigam, K.: A comparison of event models for Naive-Bayes text classification. In AAAI-98 Workshop on Learning for Text Categorization, Madison, Wisconsin (1998)
14. Schonlau, M., DuMouchel, W., Ju, W., Karr, A. F., Theus, M., Vardi, Y.: Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1) February (2001) 58-74
15. Schonlau, M., Theus, M.: Detecting masqueraders in intrusion detection based on unpopular commands. *Information Processing Letters*, 76(1-2) November (2000) 33-38
16. Wagner, R. A., Fisher, M. J.: The string-to-string correction problem. *Journal of the ACM*, Vol.21 (1974) 168-173
17. Wang, K., Stolfo, S. J.: One-class training for masquerade detection. In: 3rd ICDM Workshop on Data Mining for Computer Security (DMSEC), Florida, November (2003)

# Distributed CLP Clusters as a Security Policy Framework for Email

Saket Kaushik, Duminda Wijesekera, William Winsborough, Paul Ammann  
Center for Secure Information Systems (CSIS),  
Department of Information Systems and Software Engineering,  
George Mason University, Fairfax VA 22030.  
e-mail:{skaushik|dwijesek|wwinsbor|pammann}@gmu.edu

## Abstract

The simple mail transfer protocol (SMTP) used to transmit e-mail was designed with no security related control, resulting in it being exploited as a means to send “unwanted” email (a.k.a. “spam”). Consequently, recent extensions have concentrated on policy based management of e-mail pipeline that begins with the sender and ends with the intended receiver. As a theoretical framework to address this problem, we propose a distributed constraint logic programming (CLP) based framework in which policies are CLP modules that control message flow, and transmit messages acceptable to downstream principals. Our syntax is based on using stratified Horn clauses with constructive negation. The distributed aspect is used to enable the system to move enforcement points upstream in the message flows, as well as to enable message senders and their service providers to add headers useful for downstream actors. It is facilitated by importing and exporting predicates (cf. Maher [19]) among legal participants of the email pipeline. Accordingly our semantics consists of an appropriately tailored three-valued semantics, where stratification is used to ensure the finite termination of policy goals. For efficient implementation, we propose to materialize the policies and show that the materialization structure “faithfully models” our distributed policy.

## 1 Introduction

Due to a lack of effective control during the transmission of email messages, malicious senders are able to send a large volume of ‘unwanted’ messages, or ‘spam’, to recipients. This is clearly undesirable from the recipient’s point of view, and the scientific community has responded by inventing a plethora of techniques to fight this problem, of which prominent ones are covered in the related work section. The large number of ‘solutions’ is an evidence to the fact that there are many aspects to this problem, with each individual technique only solving a small part.

The primary hindrance to effectively utilizing different techniques for email control stems from the fact that their use introduces a risk of dropping desirable messages. Email control techniques usually require supporting documentation in the messages to aid the recipients in deciding whether to accept a message. For example, a message

with a monetary bond [18] must contain relevant information like its numeric value, currency, *etc.* In the absence of a means to convey the recipient's requirements upstream, with respect to message documentation, the senders may not be able to discover the appropriate documentation required, thereby introducing a risk of losing valuable messages. This results in recipients not utilizing the full power made available to them by these control techniques.

The second deterrent to effectively utilizing email-control techniques is an absence of a flexible means of combining them, to suit the requirements of a particular email domain. For example, currently there is no means to encode the requirement that says that "even if the sender is on the blacklist, accept the message if it is bonded with value  $> \$1.00$ ", or, "even if the bayesian filter ranks the message as spam, accept the message if the sender is a family member or a business partner". Thus, the diverse ways by which a message may be acceptable cannot be applied in practice.

In our approach, we address these two shortcomings in the following way. First, we provide a language for richer policy expressivity, that can flexibly combine the various email control techniques. We empower each principal involved in message transmission to apply acceptance criteria and thus provide an effective control mechanism. Building on a prior study [15], we also design a feedback scheme in which the senders, whose messages are not appropriately documented, are given an opportunity to do so, thus minimizing the chances of losing desirable messages.

### 1.1 Our contribution

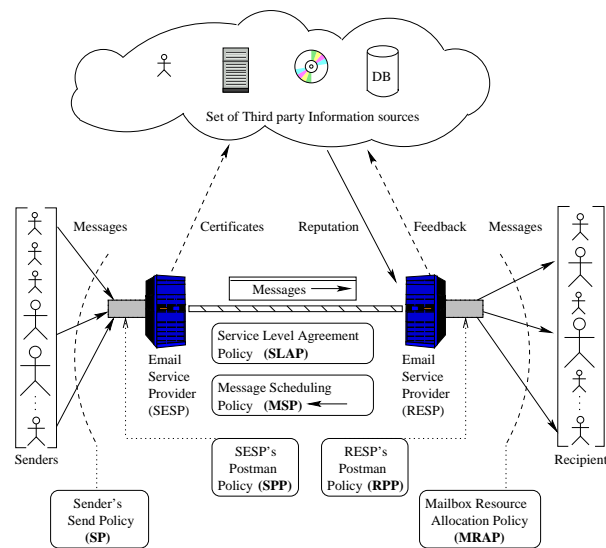
The main contributions of this paper include the following:

- A flexible policy language for expressing email control criteria including existing email control mechanisms. We identify a constraint domain, which can support acceptance criteria involving a majority of existing control mechanisms.
- A logical evaluation procedure for control decisions proposed earlier [15].
- A scheme for revising rejected messages to make them desirable to the intended recipients. It also enables recipients and other principals to effectively use precise acceptance criteria.

## 2 Policy Architecture

The heart of email is the Simple Mail Transfer Protocol (SMTP) [26], which is simply a best-effort delivery protocol with a first-come, first-served transmission policy. The protocol favors senders, ensuring reliable delivery of emails of the sender's choosing. To offset this bias we propose to empower each principal with a partial control of the email pipe, thereby allowing them to control their own participation in message transmission.

Figure 1 illustrates the principals and proposed policies in a policy-mediated email pipe. There are four principals directly involved in an email exchange. In addition to the sender and the recipient, the remaining two principals are the sender's Email Service Provider (ESP), and the recipient's ESP. In current practice ESPs help email service scale to the vast numbers of senders and recipients on the Internet. ESPs are well positioned to provide evaluations of particular messages, sender authentication, virus-scanning and their inclusion is necessary to model proposed approaches which make use of third party mechanisms, *e.g.*, reputation server [11, 23], escrow service for



**Figure 1: Principals and Policies in an Email Pipe**

Policy	Author	Provides/Expresses
SP	Sender	Instructions for revising messages
SPP	SESP	Egress filtering, revising messages, effecting delays
SLAP	RESP	Quality of service to SESP, connection filtering, delays
MSP	RESP	Message acceptance criteria, revising suggestions, delays
RPP	RESP	Ingress filtering, message delaying, prioritization
MRAP	Recipient	Individual acceptance criteria, message delaying

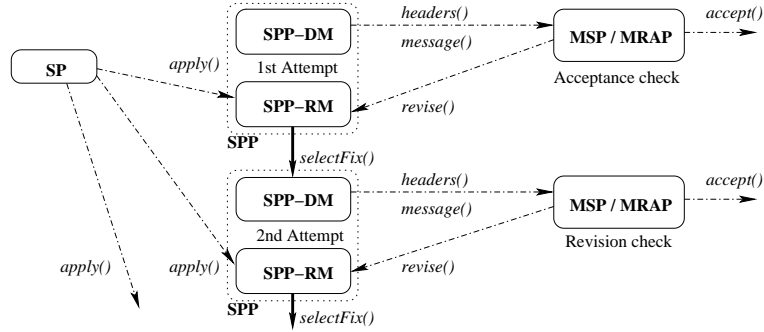
**Table 1: Control achieved through policies**

bonds [18], *etc.* It is important to note that the interests of the ESPs are separate from the interests of their clients. For example, a sending ESP with a reputation for harboring misbehaving senders might lose well-behaved customers if the Internet community as a whole treats all mail from the ESP with suspicion.

Six types of policies are identified in figure 1, based on an earlier design [15], which, we claim, are sufficient to combine a majority of email-control mechanisms. Next we briefly justify this assertion that our approach is comprehensive. In their Send Policy (SP) sender's can express their preferences in providing upgrades to messages, in order to get their emails to their destinations. SESP can provide egress filtering [5], message delaying [10] in addition to applying revisions to rejected messages using its postman policy (SPP). RESP provides connection filtering [17], and quality of service to requesting peers through the SLAP policy. Through MSP, and additionally MRAP, it can not only select messages to allow transmission, but also provide meaningful feedback for revising rejected messages. RPP, the RESP's postman policy, can prioritize delivery of important messages, message sanitization, *etc.* Finally MRAP implements individual message reception preferences like bond requirements [18], *etc.* Clearly, proposed policies give each principal sufficient control, so as to protect its interests

(see table 1).

The SP, SPP, MSP and MRAP policies are directly involved in the transmission of individual messages and their syntax is described in § 3. In broad terms, SPP and RPP are used to selectively prioritize delivery of messages, however, we do not elaborate on this function here. With SPP we focus primarily on the message delivery and revision function functions. The SPP policy can be divided into two modules: Delivery module (DM) and Revision module (RM).



**Figure 2:** Interaction of policies for message transmission

Figure 2 pictures a module network comprising of four policies: SP, SPP(= SPP-DM  $\cup$  SPP-RM), MSP and MRAP. Each module shown interacts with other modules by importing or exporting predicates. Each directed arrow identifies the source and destination of the predicate that is identified by the arrow label(s). These interactions are described next. The top-level query is ‘accept()’, which forms the interface to the module network. A message is represented as a set of facts in the SPP-DM, SPP’s delivery module. SPP-DM exports ‘message()’ and ‘header()’ predicates to MRAP and MSP policies, respectively. MSP and MRAP can ‘accept()’ a message or reject it by exporting ‘revise()’ predicate to SPP-RM, the revision module. SP’s ‘canChange()’ predicate indicates what changes can be applied. Using the ‘canChange()’ predicate, SPP-RM computes an inexpensive revision, by using the ‘selectFix()’ predicate. The revised message is then enqueued for retransmission. The ‘1st Attempt’ and ‘2nd Attempt’ labels denote initial transmission and retransmission of a revised message to emphasize the fact that the predicate communication graph is acyclic. In essence, a re-attempt can be treated as a new message.

### 3 Formal Model

#### 3.1 Syntax

**Definition 1 (Constraint domain).** *Finite Integer domain, a contiguous subset of  $\mathbb{Z}$ , denoted by  $\mathcal{FD}$  and interprets  $=, \neq, \leq, \geq, \text{minimize}(G, E)$  predicates, where  $\text{minimize}$  finds answers to the atom  $G$  (defined below in definition 6) that minimize the arithmetic expression  $E$ .*

We use elements of  $\mathcal{FD}$  to implicitly encode all alphanumeric constants, like, sender address or recipient address, etc.



**Definition 2 (Terms).** *Terms consist of only constants and variables (no function symbols), which range over  $\mathcal{FD}$ . The notation  $\vec{X}$ ,  $\vec{Y}$ , etc. is used to refer to a finite tuple, of arity 6, of terms (to conserve space).*

**Definition 3 (Headers).** *Reserved constants that refer to message attributes, called headers.*

In this paper, we consider only the headers described in table 2. Other headers can be easily added. Note that From, . . . , Auth represents 1, . . . , 6.

Email-Header	Constant	Represents
Mail From	From	1
Rcpt To	To	2
Date	Date	3
X-SESP	Sesp	4
X-Bond	Bond	5
X-Auth	Auth	6

**Table 2:** *Reserved constants for email headers*

**Definition 4 (Primitive constraint).** *A primitive constraint is of the form  $p(t_1, t_2)$  where  $p$  is a constraint relation of arity 2, taken from the list  $=, \neq, \leq, \geq$ , and minimize, and  $t_1, t_2$  are terms when  $p \in \{=, \neq, \leq, \geq\}$  and an atom (defined below in definition 6) and an arithmetic expression, respectively, when  $p$  is the minimize predicate.*

**Definition 5 (Predicates).** *Predicate symbols (described in table 3) are partitioned into following sets:*

- *Required local predicates (RL)*
- *Import-Export predicates (IE)*
- *Optional local predicates (OL)*

**Definition 6 (Atom and Literal).** *An atom is of the form  $q(t_1, \dots, t_n)$  where  $q$  is a predicate symbol or a primitive constraint and  $t_1, \dots, t_n$  are terms. A literal is an atom (i.e., a positive literal) or its negation (i.e., a negative literal).*

**Definition 7 (Clause, Fact, Rule).** *A clause is of the form  $H \leftarrow B$  where  $H$  is an atom, and  $B$  is a list of literals. A clause is called a fact if  $B = \lambda$  (empty list), and a rule otherwise.*

**Definition 8 (Policy Module).** *A policy module (or simply a module),  $M_P$ , is a set of clauses  $P_P$  with three disjoint sets of predicates: the local predicates,  $Loc_P$ , the exported predicates,  $Exp_P$ , and the imported predicates  $Imp_P$ . We require that the head of clauses in  $P_P$  be from  $Loc_P \cup Exp_P$ .*

Next we define the policy modules that we construct using the above syntax. The intuitive meanings of predicates introduced in definition 5 are presented in table 3. Readers may wish to peruse the table as they read definitions 9 – 12. In the following, we abuse the terminology at times and call an atom a predicate (to be able to discuss predicates and their arguments).

Predicate/ Arity	Description	Predicate/ Arity	Description
<b>Required Local Predicates (RL)</b>		<b>Required Local Predicates (RL) (contd.)</b>	
allow/6	defines acceptance criteria; arguments correspond to header values and delivery iteration	nHeader/2	associates header names with values
disallow/6	defines rejection criteria, same arguments as allow	<b>Import-Export Predicates (IE)</b>	
rAllow/6	acceptable revised header tuples	accept/6	combines allow & disallow, with same arguments as allow
rDisallow/6	unacceptable revised header tuples	revise/12	combines rAllow & rDisallow
Env/2	sets non-revisable header variable to the value given in the message	header/2	associates header names with original values
checkEnv/6	collects header values in the original message	content/1	associates 'content' with message provided value
fixEnv/6	collects non-revisable header values	message/7	associates header names and 'content' with values
icost/4, cost/4	cost for revising a header H with having value $R_1$ to new value $R_2$	nonFinal/1	True if header can be revised
total/13	total cost for revising headers $\vec{R}_1$ to $\vec{R}_2$	sys <sub>var</sub> /2	associates system variable 'var' to current value
selectFix/6	computes a low-cost revision to a message	prim <sub>mech</sub> /2	associates computation of 'mech' to the computed value
relay/7	defines criteria for relaying a message; arguments correspond to header values, content and delivery iteration	<b>Optional Local Predicates (OL) (Examples)</b>	
norelay/7	defines criteria for dropping a message; same arguments as relay	whitelist/1	True if argument is whitelisted
deliver/7	combines relay and norelay	blacklist/1	True if argument is blacklisted
canChange/2	permits change to header value	partner/1	True if argument is a partner

**Table 3:** Intuitive meanings of predicates

**Definition 9 (MSP policy module).** An MSP policy module  $M_{MSP}$ , is given by  $\langle Imp_{MSP}, Exp_{MSP}, Loc_{MSP}, P_{MSP} \rangle$ , in which  $Imp_{MSP} = \{header, nonFinal, sys_{var_i}, prim_{Mech_j}\}$ ,  $Exp_{MSP} = \{accept, revise\}$ ,  $Loc_{MSP} = \{allow, disallow, rAllow, rDisallow, checkEnv, fixEnv, Env\} \cup OL$  predicates and  $P_{MSP}$ , a set of facts and stratified [22] rules, which has four strata:

- **Stratum 0:** Definitions of RL predicates:  $checkEnv(\vec{X})$ ,  $fixEnv(\vec{X})$  and  $Env(X, Y)$ , where  $\vec{X} = X_{From}, \dots, X_{Auth}$ . These are as follows:

$$\begin{aligned}
 checkEnv(\vec{X}) \leftarrow & \quad header(From, X_{From}), header(To, X_{To}), \\
 & \quad header(Date, X_{Date}), header(Sesp, X_{Sesp}), \\
 & \quad header(Bond, X_{Bond}), header(Auth, X_{Auth}). \quad (1)
 \end{aligned}$$

$$fixEnv(\vec{X}) \leftarrow Env(From, X_{From}), \dots, Env(Auth, X_{Auth}) \quad (2)$$

$$Env(H, X_H) \leftarrow nonfinal(H) \quad (3)$$

$$Env(H, X_H) \leftarrow header(H, X_H) \quad (4)$$

- **Stratum 1:** Definitions of all OL predicates
- **Stratum 2:** Definition of following RL predicates:  $allow(\vec{X})$ ,  $disallow(\vec{X})$ ,  $rAllow(\vec{X})$  and  $rDisallow(\vec{X})$ . In addition to Strata 0,1 predicates, most of IE predicates can be used in the body of the defined rules, except  $content(X)$ ,  $message(\vec{X})$ , and  $prim_{mech}(X, Y)$

predicates. As a further restriction, allow and disallow predicates cannot use fixEnv or Env predicates. rAllow and rDisallow are constructed from allow and disallow predicate definitions in the following manner. For each allow (resp., disallow) rule, a rule with rAllow head (resp., rDisallow) is generated with checkEnv replaced by fixEnv.

- **Stratum 3:** RL predicate accept, defined as,  $\text{accept}(\vec{X}) \leftarrow \text{allow}(\vec{X}), \neg \text{disallow}(\vec{X})$ ;
- **Stratum 4:** RL predicate revise, defined as,  $\text{revise}(\vec{X}, \vec{Y}) \leftarrow \text{checkEnv}(\vec{X}), \neg \text{accept}(\vec{X}), \text{rAllow}(\vec{Y}), \neg \text{rDisallow}(\vec{Y})$ .

Note that constraints do not occur in the rules (1–4) as Env, checkEnv, and fixEnv predicates are used to set up the environment, i.e., the values of headers provided in the message. Constraints on variables are expressed in the rest of the policy. Negation is used in a very limited way, such as, in the definitions of system defined predicates, like, accept, deliver, etc., or with allow and disallow, etc. Negation can be used in OL predicates, provided some conditions are met. Primarily, OL predicate definitions must be stratified according to the following restrictions (i) can positively refer to atoms whose predicate symbols are defined in the same or lower OL strata, and atoms in Stratum 0 (ii) can negatively refer to atoms whose predicate symbols are defined in a lower OL strata and atoms in Stratum 0. The OL strata forms substrata of Stratum 1.

**Definition 10 (MRAP policy module).** An MRAP policy module,  $M_{MRAP}$  is given by  $\langle \text{Imp}_{MRAP}, \text{Exp}_{MRAP}, \text{Loc}_{MRAP}, P_{MRAP} \rangle$ , in which  $\text{Imp}_{MRAP} = \{ \text{content}, \text{message} \} \cup \text{Imp}_{MSP}$ ,  $\text{Exp}_{MRAP} = \{ \text{accept}_a, \text{revise}_a \}$ ,  $\text{Loc}_{MRAP} = \{ \text{allow}_a, \text{disallow}_a, \text{rAllow}_a, \text{rDisallow}_a, \text{checkEnv}_a, \text{fixEnv}_a, \text{Env}_a \}$  and  $P_{MRAP}$  is a set of stratified clauses, with the following strata (all Loc and Exp predicates have been subscripted to prevent name collisions with MSP policy):

- **Stratum 0:** Definitions of RL predicates  $\text{checkEnv}_a(\vec{X})$ ,  $\text{fixEnv}_a(\vec{X})$  and  $\text{Env}_a(X, Y)$ , as defined in the MSP stratum 0.
- **Stratum 1:** Definitions of OL predicates.
- **Stratum 2:** Definitions of the following RL predicates:  $\text{allow}_a(\vec{X})$ ,  $\text{disallow}_a(\vec{X})$ ,  $\text{rAllow}_a(\vec{X})$  and  $\text{rDisallow}_a(\vec{X})$ . All IE predicates can be used in the body of the defined rules. MSP restrictions on use of  $\text{fixEnv}_a$  apply.  $\text{rAllow}_a$ ,  $\text{rDisallow}_a$  are constructed as shown in MSP stratum 2.
- **Strata 3 and 4:** RL predicate  $\text{accept}_a(\vec{X})$  (Stratum 3) and  $\text{revise}_a(\vec{X}, \vec{Y})$  (Stratum 4), defined similar to the corresponding predicate definitions in MSP strata 3 and 4.

**Definition 11 (SP policy module).** SP policy module,  $M_{SP} = \langle \text{Imp}_{SP}, \text{Exp}_{SP}, \text{Loc}_{SP}, P_{SP} \rangle$ , in which  $\text{Imp}_{SP} = \emptyset$ ,  $\text{Exp}_{SP} = \{ \text{canChange} \}$ ,  $\text{Loc}_{SP} = \emptyset$  and  $P_{SP}$ , is a set of rules or facts that define  $\text{canChange}(X, Y)$  clauses.

**Definition 12 (SPP).** SPP consists of two modules:

**SPP-DM: Delivery module** Delivery module,  $M_{SPP-DM} = \langle \text{Imp}_{SPP-DM}, \text{Exp}_{SPP-DM}, \text{Loc}_{SPP-DM}, P_{SPP-DM} \rangle$ , in which  $\text{Imp}_{SPP-DM} = \{ \text{canChange} \}$ ,  $\text{Exp}_{SPP-DM} = \{ \text{header}, \text{content}, \text{message}, \text{nonFinal} \}$ ,  $\text{Loc}_{SPP-DM} = \{ \text{relay}, \text{norelay}, \text{deliver} \}$  and  $P_{SPP-DM}$ , consists of following strata of rules:

- **Stratum 0:** Facts -  $\text{header}(X, Y)$ ,  $\text{content}(X)$  and  $\text{nonFinal}(X)$  and definition of  $\text{message}(\vec{X}, C)$  predicate:  $\text{message}(\vec{X}, C) \leftarrow \text{header}(\text{From}, X_{\text{From}}, \dots, \text{header}(\text{Auth}, X_{\text{Auth}}), \text{content}(C))$ ; where  $\vec{X} = X_{\text{From}}, \dots, X_{\text{Auth}}$ .
- **Stratum 1:** Definition of RL predicates  $\text{relay}(\vec{X}, C)$  and  $\text{norelay}(\vec{X}, C)$ . Definitions can use IE predicates and lower strata predicates in the body.

- **Stratum 2:** Definition of SP predicate  $\text{deliver}(\vec{X}, C)$ , defined as,  $\text{deliver}(\vec{X}, C) \leftarrow \text{relay}(\vec{X}, C), \neg \text{norelay}(\vec{X}, C)$

**SPP-RM Revision module** Revision module,  $M_{\text{SPP-RM}} = \langle \text{Imp}_{\text{SPP-RM}}, \text{Exp}_{\text{SPP-RM}}, \text{Loc}_{\text{SPP-RM}}, P_{\text{SPP-RM}} \rangle$ , in which  $\text{Imp}_{\text{SPP-RM}} = \{\text{revise}, \text{revise}_a\}$ ,  $\text{Exp}_{\text{SPP-RM}} = \emptyset$ ,  $\text{Loc}_{\text{SPP-RM}} = \{\text{cost}, \text{icost}, \text{total}, \text{selectFix}\}$  and  $P_{\text{SPP-RM}}$ , a set of rules that define  $\text{selectFix}(\vec{X})$ ,  $\text{total}(\vec{X}, \vec{Y}, Z)$ ,  $\text{cost}(H, X, Y, C)$  and  $\text{icost}(H, X, Y, C)$  predicates. The  $\text{icost}(H, X, Y, C)$  predicate is defined by a collection of facts that define a cost  $C$  to change header value  $X$  to  $Y$ . The predicates  $\text{nHeader}$  and  $\text{cost}$  are defined as ( $\text{maxInt}$  is the largest integer in  $\mathcal{FD}$ ):

$$\text{cost}(H, X, Y, C) \leftarrow \text{canChange}(H, Y), \text{icost}(H, X, Y, C) \quad (5)$$

$$\text{cost}(H, X, Y, \text{maxInt}) \leftarrow \neg \text{canChange}(H, Y) \quad (6)$$

$$\text{nHeader}(H, X_H) \leftarrow \text{selectFix}(X_{\text{From}}, \dots, X_H, \dots, X_{\text{Auth}}) \quad (7)$$

$$\text{nHeader}(H, X_H) \leftarrow \neg \text{selectFix}(X_{\text{From}}, \dots, X_H, \dots, X_{\text{Auth}}), \text{nHeader}(H, X_H) \quad (8)$$

**Definition 13 (Supporting modules).** Supporting policy modules include the system module, a four tuple:  $\langle \emptyset, \{\text{sysvar}_i\}, \emptyset, F_s \rangle$ , and primitive mechanisms, which are given by following forms of 4 tuples:  $\langle \{\text{header}(h, X_h)\}, \{\text{prim}_{\text{mech}_i}\}, \emptyset, F_p \rangle$ .

The sets of clauses ( $F_s$ , etc.) in the supporting modules are essentially a set of facts. In practice, these are constructed at evaluation time based on system environment conditions. The number of supporting modules varies depending on the predicates used in the MSP and the MRAP modules. We use a set  $\text{SM}_n$ , to represent *all* the supporting modules whose predicates are imported by the MSP and the MRAP modules.

**Example 1 (MSP or MRAP policy module).** Consider the Stratum 3:

$$\text{allow}(\vec{X}) \leftarrow \text{checkEnv}(\vec{X}), \text{whitelist}(X_{\text{From}}). \quad (9)$$

$$\text{disallow}(\vec{X}) \leftarrow \text{checkEnv}(\vec{X}), \text{blacklist}(X_{\text{From}}). \quad (10)$$

$$\text{allow}(\vec{X}) \leftarrow \text{checkEnv}(\vec{X}), X_{\text{Auth}} = \text{PKI}. \quad (11)$$

Rule 9 says that a message is acceptable if the sender is in the recipient's whitelist. Rule 10 says that a message is unacceptable if the sender is found to be on the recipient's blacklist. Rule 11 says that a message is acceptable whenever the message has been strongly authenticated. Together, the rules allow emails from senders who are whitelisted or who strongly authenticate their messages, and block messages from blacklisted senders.

**Example 2 (SP policy module).**

$$\text{canChange}(\text{Bond}, C) \leftarrow C < 5 \quad (12)$$

$$\text{canChange}(\text{Auth}, \text{'PKI'}) \leftarrow \quad (13)$$

Rule 12 says that if required, a bond of value less than 5 can be applied to the outgoing message. Similarly, fact 13 says that sender's private key can be used to add digital signatures or other forms of PKI based authentication.

**Example 3 (SPP).** A simple delivery module ( $\text{SPP-DM}$ ):

$$\text{relay}(\vec{X}, C) \leftarrow \text{message}(\vec{X}, C), \neg \text{prim}_{\text{Nortan}}(C) \quad (14)$$

$$\text{relay}(\vec{X}, C) \leftarrow \text{message}(\vec{X}, C), \neg \text{prim}_{\text{crm2}}(C, X), X < 0.3 \quad (15)$$

Rule 14 says that if the message is found free of any virus, it can be delivered. Rule 15 states a message ranking low on the spam filter, can be delivered. Together the rules require a message to undergo either a virus-scan or a filtering process to be allowed delivery.

A simple revision module (RM):

$$\begin{aligned} \text{total}(\vec{R}_1, \vec{R}_2, \vec{C}) &\leftarrow \text{revise}(\vec{R}_1, \vec{R}_2), \text{cost}(\text{From}, R_{1,\text{From}}, R_{2,\text{From}}, C_1), \dots, \\ &\quad \text{cost}(\text{Auth}, R_{1,\text{Auth}}, R_{2,\text{Auth}}, C_6) \end{aligned} \quad (16)$$

$$\text{selectFix}(\vec{R}_2) \leftarrow \text{minimize}(\text{total}(\vec{R}_1, \vec{R}_2, \vec{C}), C_1 + \dots + C_6). \quad (17)$$

Rule 16 shows how to calculate the costs to revise a message using the cost predicate and vectors  $\vec{R}_1, \vec{R}_2$  whose individual elements are  $R_{1,\text{From}}, R_{2,\text{From}}$ , etc. and  $C_j$  is the cost for changing  $R_{1,j}$  to  $R_{2,j}$ . Also,  $\vec{C} = C_1, \dots, C_6$ . Rule 17 calculates the minimum cost change by minimizing the total cost of revision.

**Definition 14 (System of policy modules).** A system of policy modules,  $\Gamma$ , is given by a finite set of policy modules, indexed by  $I$ . Thus  $M_i$  ranges over modules in  $\Gamma$  for  $i \in I$ . For any  $i, j \in I$ ,  $i \neq j$ , we have  $\text{Loc}_i \cap \text{Loc}_j = \emptyset \wedge \text{Exp}_i \cap \text{Exp}_j = \emptyset$ . For such  $i, j$  we write  $i \sqsubset_\Gamma j$  if  $\text{Exp}_i \cap \text{Imp}_j \neq \emptyset$ . Letting  $\sqsubset_\Gamma^*$  denote the transitive closure of  $\sqsubset_\Gamma$ , we require that  $\forall i \in I, i \not\sqsubset_\Gamma^* i$  (irreflexivity), i.e., the relation  $\sqsubset_\Gamma^*$  is a partial order.

**Definition 15 (Complete system of policy modules).** A system of policy modules is a complete system of policy modules if  $\forall i \in I \text{ Imp}_i \subset \bigcup_{j \in I \wedge j \neq i} \text{Exp}_j$

**Theorem 1.** A set of policy modules consisting of one of each policy module types: SP, SPP-DM, SPP-RM, MSP, MRAP and  $SM_n$  forms a complete system of policy modules.

**Proof:** We give an informal proof to show that the set of policy modules  $\Gamma_1 = \{M_{SP}, M_{SPP-DM}, M_{SPP-RM}, M_{MSP}, M_{MRAP}\} \cup SM_n$  satisfies all the criteria to be a complete system of policy modules. By inspection, following properties can be ascertained in a straightforward manner. Firstly, for each pair of distinct policy modules  $M_i$  and  $M_j$  ( $i, j$  range over an index  $I_1$  of  $\Gamma_1$ ),  $\text{Loc}_i \cap \text{Loc}_j = \emptyset \wedge \text{Exp}_i \cap \text{Exp}_j = \emptyset$  holds. Secondly, the relation  $\sqsubset_{\Gamma_1}$  is irreflexive and transitive (i.e., the condition  $\forall i \in I_1, \nexists k_1, \dots, k_n \in I_1$  such that  $i \sqsubset_{\Gamma_1}^* i$ , can be easily verified). Finally, to see the completeness property, we construct a set  $\bigcup_{j \in I_1} \text{Imp}_j$  and verify that  $\bigcup_{j \in I_1} \text{Imp}_j \subset \bigcup_{k \in I_1} \text{Exp}_k$ .  $\square$

**Notation 1 (System of email policy modules).** A complete system of email policy modules is represented by  $\Gamma_{em}$ , which is given by the set  $\{M_i, SM_j \mid i \in I_1 \text{ and } j \in I_2\}$ , where  $I_1 = \{SP, SPP-DM, SPP-RM, MSP, MRAP\}$  and  $I_2 = \{1, \dots, k\}$ ,  $SM_1, \dots, SM_k \in SM_n$  and each  $M_i, SM_j$  satisfies the appropriate definitions, 9 – 13, above. The combination of these indices is represented by  $I_{em} = I_1 \cup I_2$ .

**Theorem 2 (Stratification [22]).**  $\forall i \in I$ , the sets  $P_i$  and  $\bigcup_i P_i$  are stratified.

**Proof:** Stratification for MSP may be obtained through the following level mapping [22, 2]  $\ell$ .  $\ell$  assigns all imported predicates header, nonFinal,  $\text{prim}_{Mech_j}$  and  $\text{syst}_{Var_k}$ , to the level 1. Predicates checkEnv, fixEnv and Env are assigned the level 2; 3 is assigned to all OL predicates; 4 is assigned to the RL predicates defined in Stratum 2 of the policy; 5 is assigned to accept and 6 to revise.

MRAP Stratification is obtained in a similar manner as above, with the addition of message, content,  $\text{prim}_{\text{Nortan}}$  and  $\text{prim}_{\text{crm1}}$  to level 1.

Following level mapping,  $\ell$ , assigns strata to SPP predicates as follows: 0 is assigned to header, content, nonFinal, and icost. Level 1 is assigned cost. Level 6 is assigned to the IE predicates revise,  $\text{revise}_a$  and RL predicates total, selectFix; nHeader is assigned the level 7; relay, norelay to 8 and deliver is assigned to level 9.

The level mapping for SP is trivial, with canChange being assigned the level 0. Similarly, all predicates defined in  $\text{SM}_n$  modules are assigned to 0. The level mapping for  $\bigcup_i P_i$  is simply the union of the level mappings for individual policies.  $\square$

### 3.2 Semantics

We use a three-valued semantics, called Kunen-Fitting (or Kripke-Kleene) [13] semantics, for interpreting our normal CLP programs. We use constructive negation [3] as proposed by Fages [12]. We first repeat some standard definitions as they appear in [13] and are repeated in [31].

**Definition 16 ( $P^*$ ,  $T_P$  and  $\Phi_P \uparrow$  operators).** Suppose  $P$  is a policy module, and let  $P^*$  be all ground instances of clauses in  $P$ . We now define two and three valued truth lattices to be  $2 = \langle \{T, F\}, <_2 \rangle$  and  $3 = \langle \{T, F, \perp\}, <_3 \rangle$  respectively, where  $T, F$  and  $\perp$  are taken to mean true, false and unknown truth values. Partial orderings  $<_2$  and  $<_3$  satisfy as  $F <_2 T$  and  $\perp <_3 T, \perp <_3 F$  respectively. A mapping  $V$  from the herbrand base of  $P$  to 2 or (respectively 3) is said to be a two-valued (respectively a three-valued) valuation of  $P$ . Any valuation is naturally extended to negative literals according to the following interpretation of negation:  $\neg T = F$ ,  $\neg F = T$  and  $\neg \perp = \perp$ . Also,  $\alpha \vee \beta = T$  if  $\alpha = T$  or  $\beta = T$ ;  $\alpha \vee \beta = F$  if  $\alpha = F$  and  $\beta = F$ ; and  $\alpha \vee \beta = \perp$  otherwise.  $\vee$  extends pointwise to valuations. Given a valuations  $V_l$  and  $V_i$ , the two and three valued immediate consequence operators  $T_P^{V_i}(V_l, V_i)$  and  $\Phi_P^{V_i}(V_l, V_i)$  are defined as follows:  
 $T_P^{V_i}(V_l, V_i): T_P^{V_i}(V_l, V_i) = W$  is defined as

- $W(H) = T$  if there is a ground clause  $H \leftarrow B$  in  $P^*$  such that  $V_i(B_k) = T$  for all  $B_k \in B$  constructed using a predicate in  $\text{Imp}_P$  and  $V_l(B_m) = T$  for all  $B_m \in B$  constructed using predicates not in  $\text{Imp}_P$ .
- $W(H) = F$  otherwise.

$\Phi_P^{V_i}(V_l, V_i): \Phi_P^{V_i}(V_l, V_i) = W$  is defined as

- $W(H) = T$  if there is a ground clause  $H \leftarrow B$  in  $P^*$  such that  $V_i(B_k) = T$  for all  $B_k \in B$  constructed using a predicate in  $\text{Imp}_P$  and  $V_l(B_m) = T$  for all  $B_m \in B$  constructed using predicates not in  $\text{Imp}_P$ .
- $W(H) = F$  if for every ground clause  $H \leftarrow B$  in  $P^*$ ,  $V_i(B_k) = F$  for some  $B_k \in B$  constructed using a predicate in  $\text{Imp}_P$  or  $V_l(B_m) = F$  for some  $B_m \in B$  constructed using predicates not in  $\text{Imp}_P$ .
- $W(H) = \perp$  otherwise.

Now we define bottom-up semantics for both  $T_P$  and  $\Phi_P$ , where  $\Psi$  stands for either of them in the following:

- $\Psi_P^{V_i} \uparrow (0) = V_{\text{false}}$ , where  $V_{\text{false}}$  assigns  $F$  (false) to all instantiated atoms.
- $\Psi_P^{V_i} \uparrow (\alpha + 1) = \Psi_P^{V_i}(\Psi_P^{V_i} \uparrow (\alpha), V_i)$  for every successor ordinal  $\alpha$ .
- $\Psi_P^{V_i} \uparrow (\alpha) = \bigvee_{\beta < \alpha} (\Psi_P^{V_i} \uparrow (\beta))$  for limit ordinal  $\alpha$ .

**Definition 17 (Bottom-up semantics).** Let  $P_i \in \Gamma$  be a policy module and  $\Phi$  be the three valued consequence operator as defined above. We let  $\Phi_P^{V_i} \uparrow (\omega)$  be the semantics of  $P$ .

**Definition 18 (Projection operator).** Given a valuation  $V$  and a set of predicates  $P$  such that  $V$  is defined over atoms constructed using predicates in  $P$  and possibly some other predicates, projection  $V|_P$  is the valuation defined over only atoms constructed using predicates in  $P$  and having the same value as  $V$  on those atoms.

### 3.2.1 An example MSP Evaluation

Next we present a simple MSP policy module evaluation in an example.

**Example 4.** Policy module evaluation without feedback

Consider a message with only three headers:

Mail From: sender@abc.com (final); Rcpt To: recipient@xyz.com (final); X-Auth: 'Password'(final);

The Keyword 'final' in the headers indicates that they cannot be revised. More details on 'final' are given in section 3.2.2. We show the evaluation of the module described in example 1 next. We assume that the sender does not belong to either the whitelist or the blacklist. In the  $\Phi_{P_{MSP}}^{V_i}$  computation, presented in table 4, we omit the '@abc.com' part in sender address, and predicates like  $rAllow$ ,  $rDisallow$ ,  $revise$  and  $fixEnv$ . We use meta-variable  $\vec{x}$  to refer to a tuple of constants, i.e., header values. Also, we only show the whitelist/blacklist predicate instances for sender@abc.com.

Ordinal	$W(H)=T$	$W(H)=F$
1	{cHeader(Auth,'Password'), cHeader(From, 'sender'), ... }	{ whitelist('sender'), blacklist('sender'), ..., nonFinal(Auth), nonFinal(From), nonFinal(To) }
2	{ cHeader(Auth,'Password'), cHeader(From, 'sender'), ..., checkEnv( $\vec{x}$ ) }	{ whitelist('sender'), blacklist('sender'), ..., nonFinal(Auth), ... }
3	{ cHeader(Auth,'Password'), cHeader(From, 'sender'), ..., checkEnv( $\vec{x}$ ) }	{ allow( $\vec{x}$ ), disallow( $\vec{x}$ ), whitelist('sender'), blacklist('sender'), ..., nonFinal(Auth), ... }
4	{ cHeader(Auth,'Password'), cHeader(From, 'sender'), ..., checkEnv( $\vec{x}$ ) }	{ accept( $\vec{x}$ ), allow( $\vec{x}$ ), disallow( $\vec{x}$ ), whitelist('sender'), blacklist('sender'), ..., nonFinal(Auth), ... }

**Table 4:**  $\Phi_{P_{MSP}}^{V_i}$  calculation for message acceptance

The fixpoint is reached at ordinal 4. Since  $\Phi_{P_{MSP}}^{V_i} \uparrow (4)$  cannot prove *accept*, the message is rejected. The root cause is that the authentication provided in the message is inconsistent with that required in the policy.

### 3.2.2 Message revision

The messages rejected, as in example 4, may be desirable to the recipient, and should not be dropped without letting the sender know why they were dropped. If this information is provided to the SPP, it can appropriately revise an initially rejected message and, thus, successfully deliver it. To do so, we require a minor change in how messages are documented. Message headers are required to include a qualifier, namely, 'final', for every header that cannot be revised. This indicates to the MSP or the MRAP which headers can be revised. This documentation is captured by the nonFinal(H) atom. Using the fixEnv( $\vec{X}$ ) atom in the definitions of constructed predicates  $rAllow(\vec{X})$  and  $rDisallow(\vec{X})$ , policy-desired constraints can be captured in *revise* predicate. Ground instances of *revise* are exported by the evaluating module to indicate to the SPP, the desired documentation in a rejected message.

**Example 5. Policy module evaluation with message revision**

Message rejected in example 4 can be revised if marked as follows:

*Mail From: sender@abc.com (final); Rcpt To: recipient@xyz.com (final); X-Auth: 'Password';*

Here the SESP indicates that authentication can be upgraded, if required. Next we show the evaluation of the policy rules to revise this rejected message. The  $\Phi_{P_{MSP}}^{V_i}$  computation is presented in table 5. The fixpoint is reached at ordinal 5 and  $\Phi_{P_{MSP}}^{V_i} \uparrow (5)$  proves a  $\text{revise}(\vec{X}, \vec{Y})$  atom.

Ordinal	$W(H)=T$	$W(H)=F$
1	{nonFinal(Auth), cHeader(From,'sender'), cHeader(To,'recipient'), cHeader(Auth,'Password')}	{nonFinal(From), nonFinal(To), whitelist(sender), blacklist(sender), ... }
2	{nonFinal(Auth), cHeader(From,'sender'), ..., checkEnv( $\vec{x}$ ), Env(From,'sender'), Env(To,'recipient'), Env(Auth,...)}	{nonFinal(From), nonFinal(To), whitelist(sender), blacklist(sender), ... }
3	{nonFinal(Auth), cHeader(From,'sender'), ..., checkEnv( $\vec{x}$ ), Env(From,'sender'), ..., fixEnv( $\vec{x}$ )}	{allow( $\vec{x}$ ), disallow( $\vec{x}$ ), nonFinal(From), nonFinal(To), whitelist(sender), blacklist(sender), ... }
4	{nonFinal(Auth), cHeader(From,'sender'), ..., checkEnv( $\vec{x}$ ), Env(From,'sender'), ..., fixEnv( $\vec{x}$ ), rAllow( $\vec{x}$ )}	{allow( $\vec{x}$ ), disallow( $\vec{x}$ ), nonFinal(From), nonFinal(To), whitelist(sender), blacklist(sender), ..., accept( $\vec{x}$ ), rDisallow( $\vec{x}$ )}
5	{nonFinal(Auth), cHeader(From,'sender'), ..., checkEnv( $\vec{x}$ ), Env(From,'sender'), ..., fixEnv( $\vec{x}$ ), rAllow( $\vec{x}$ ), revise( $\vec{x}, \vec{y}$ )}	{allow( $\vec{x}$ ), disallow( $\vec{x}$ ), nonFinal(From), nonFinal(To), whitelist(sender), blacklist(sender), ..., accept( $\vec{x}$ ), rDisallow( $\vec{x}$ )}

**Table 5:**  $\Phi_{P_{MSP}}^{V_i}$  calculation for message revision

## 4 Materialization Structure

Section 3 establishes stratification [22] of each local policy module and their union. This essentially allows each delivery attempt being treated as a new message delivery. Thus, for a single message delivery attempt, the module graph is acyclic, as required in [19]. There can be many strategies for exporting predicates, like, for instance, ‘one-at-a-time’ transmission or ‘all-together’ transmission strategies. In our view, the second approach can be more efficient in reducing communication overheads, and hence we propose materializing exported predicates to support this strategy.

**Definition 19 (Materialization Structure  $\mathcal{MS}$ ).** The materialization structure,  $\mathcal{MS}_i$ , of a module  $M_i$  is a valuation over atoms constructed using the  $\text{Exp}_i$  predicates.

**Definition 20 (Correctness).** Given a system of policy modules  $\Gamma$  indexed by  $I$ , a corresponding set of materialization structures also indexed by  $I$ , and ranged over by  $\mathcal{MS}_i$ , is correct if  $\forall i \in I \mathcal{MS}_i = \Phi_{P_i}^{V_i} \uparrow (\omega) \mid_{\text{Exp}_i}$ , where  $V_i = \bigcup_{j \sqsubset_{\Gamma} i} (\mathcal{MS}_j \mid_{\text{Imp}_i})$  (Here we are viewing the function  $\mathcal{MS}_j$  as sets of pairs. In this way combine these functions whose domains are disjoint).

**Theorem 3 (Faithfulness and Adequacy).** Given a complete system of policy modules  $\Gamma$ , indexed by  $I$ , and a corresponding set of materialization structures also indexed by  $I$  and ranged over by  $\mathcal{MS}_i$ ,  $\bigcup_{i \in I} \mathcal{MS}_i = \Phi_P^{\emptyset} \uparrow (\omega) \mid_{\text{Exp}}$ , where  $P = \bigcup_{j \in I} P_j$ ,  $\text{Exp} = \bigcup_{j \in I} \text{Exp}_j$ .

**Proof Strategy:** The proof strategy of using induction over  $i$  would give the required proof.  $\square$



Due to theorem 1, theorem 3 applies to the complete system of email policies, which is significant for our application. The communication of imported-exported predicates between modules is proposed through a materialization of individual (local) modules. The evaluation of the next module, *i.e.*, the module whose predecessor(s) has (have) completed its (their) evaluation(s), depends upon the inputs from predecessors. The significance of theorem 3 lies in the fact that it shows the correctness of the ‘relative’ (local) evaluation scheme with respect to an evaluation scheme with the possession of global knowledge.

## 5 Related work

*Content based filtering and text categorization solutions*, are current best response to spam. A filter attempts to parse message content and use pattern matching with text categorization to identify undesirable mail. Vast advances have been made in this field, For example, in [4, 20, 14] *etc.*, which claim to catch a majority of ‘junk’ mail. However, one of its main drawbacks is the problem of false negatives and false positives. Since ‘all’ unwanted messages cannot be caught, spammers simply increase the volume of spam to achieve the same yield as before. Statistics show that spam is now more than half the bandwidth of all mail messages sent and the cost of handling these messages and transmitting them close to the destination has to be borne by the infrastructure. Hence, we provide a pro-active mechanism in this paper to stop unwanted messages closer to the source.

*Economic solutions* address the question of who pays for the cost of dealing with spam. Allman [1] advocates developing techniques to share these costs between senders and recipients, as a means to reduce spam incidences pro-actively. Works like [18, 6, 9] provide a basis of such an approach. However, what is lacking is the translation of these techniques to the SMTP. The authors do not show how their schemes can be incorporated incrementally in a backward-compatible manner to the email delivery protocol. In our approach, which is based on [15], these solutions can be easily incorporated, by expressing the (monetary) preferences in an MSP or MRAP policy and can be combined with other anti-spam techniques. SLAP policy can slow down spammers, similar to the computational and memory bound cost mechanisms above.

*Reputation and trust based solutions* are dependent on establishing identity of the sender, which is a significant problem in email. Based on identity, whitelists and blacklists are defined, which can accept or reject messages respectively. Identity spoofing mitigation and reputation based systems have been explored in various works like [30, 21, 11], however, Friedham *et al.* in [24] and Watson in [32] show that spoofing problem is not easily solved. In any case, requirements like blacklists, whitelists, DNS-specific requirements, trust-based requirements can be easily expressed in our policies and combined together in an arbitrary fashion.

*Network based solutions* use network characteristics to discover and thwart spam. Sender policy framework [25], reverse DNS checks [7] help solve spoofing by verifying domain name against IP-address and fit well in our SLAP policy. Realtime Blackhole Lists (RBLs) [23] serve as a reputation service that reports spammer IP addresses, which would enhance the MRAP policies. Message reputations schemes [8, 28, 27] are used to identify ‘bulk mail’ and would fit well in our policies. Clayton’s extrusion detection technique [5] is a novel idea, easily supported in our framework through SPP

policies. Li's TCP damping approach [17] can be a part of our SLAP policy. Our policies can combine low level network attributes with high level attributes like spam index, bond amount *etc.*, powerful anti-spam mechanisms can be constructed.

## 6 Conclusion

As noted by Leiba *et al.* [16], protection against receiving undesirable messages is hard to obtain, unless, we can combine different types of available techniques that stop such messages at their own level. However, existing literature shows that, so far, flexible and extensible combinations of techniques have not been successfully provided. To this end, we propose a policy-based approach to flexibly combine available email control techniques, and provide a CLP-based mechanism to evaluate such policies. By incorporating minor changes into the way messages are constructed, we are able to extend our mechanism to provide constructive feedback to the senders, such that, desirable messages that do not meet evaluation policy requirements, can be 'revised' and hence make it to their destination. This approach is a vast improvement over the current practice, where desirable messages are killed if they get 'flagged' by spam filters. Thus, by minimizing the risk of dropping desirable messages, our scheme enables policy authors to effectively use precise criteria for message acceptance, a goal that could not be realized in the current message transmission framework. Of course, this risks in leaking private information, such as, content of ESP maintained blacklists, whitelists *etc.*, but policy communications can be sanitized. We can handle this requirement efficiently in our framework, but it is out of scope of current work and we plan to treat it elsewhere.

## Acknowledgement

We thank the anonymous reviewers for their valuable comments on improving the presentation of this paper.

## References

- [1] E. Allman. The economics of spam. <http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=108>.
- [2] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming*, pages 89–148, 1988.
- [3] D. Chan. Constructive negation based on the completed databases. In *International conference on logic programming (ICLP)*, pages 111–125, 1988.
- [4] S. Chhabra, W. S. Yezauris, and C. Siefkes. Spam filtering using a markov random field model with variable weighting schemas. In *ICDM'04: Fourth IEEE International Conference on Data Mining*, To appear 2004.
- [5] R. Clayton. Stopping spam by extrusion detection. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [6] R. Dai and K. Li. Shall we stop all unsolicited email messages? In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [7] Danisch.de: Defense against spam and E-Mail forgery. <http://www.danisch.de/work/security/antispam.html>.
- [8] Distributed Checksum clearinghouse. <http://rhyolite.com/anti-spam/dcc/>.
- [9] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO'03: Advances in cryptology*, 2003.

- [10] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO'92: Advances in cryptology*, 1992.
- [11] Email Service Provider Coalition. Project Lumos. <http://www.networkadvertising.org/espc/lumos.white.paper.asp>, Sept 2003.
- [12] F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32/2, 1997.
- [13] M. C. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2/4:295–312, 1985.
- [14] A. Gray and M. Haahr. Personalized collaborative spam filtering. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [15] S. Kaushik, P. Ammann, D. Wijesekera, W. Winsborough, and R. Ritchey. A policy driven approach to email services. In *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, New York, June 2004.
- [16] B. Leiba and N. Borenstein. A multifaceted approach to spam reduction. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [17] K. Li, C. Pu, and M. Ahamad. Resisting spam delivery by tcp damping. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [18] T. Loder, M. V. Alstyne, and R. Wash. An economic solution to the spam problem. In *5th ACM conference on Electronic Commerce*, 2004.
- [19] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
- [20] E. Michalakos, I. Androustopoulos, G. Paliouras, G. Sakkis, and P. Stamatopoulos. Filtron: A learning-based anti-spam filter. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.
- [21] M. Naor. Verification of a human in the loop or identification via the turing test. <http://www.wisdom.weizmann.ac.il/naor/PAPERS/human.abs.html>, 1996.
- [22] T. C. Przymusiński. On the declarative semantics of stratified deductive databases and logic programs. *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1987.
- [23] Realtime Blackhole List. <http://www.kelkea.com/>.
- [24] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. Reputation systems. *Communications of the ACM*, 43(12):45–48, December 2000.
- [25] Sender Policy Framework. <http://spf.pobox.com>.
- [26] Simple Mail Transfer Protocol. RFC 2821, Apr 2001.
- [27] Spam URI Realtime Blocklist. <http://surbl.org/>.
- [28] SpamNet. <http://www.cloudmark.com>.
- [29] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118/1:12–33, 1995.
- [30] T. Tomkins and D. Handley. Giving email back to the users: using digital signatures to solve the spam problem. *First Monday*, 8(9), September 2003.
- [31] L. Wang, D. Wijesekera, and S. Jajodia. A logic based framework for attribute based access control. In *2nd ACM Workshop on Formal Methods in Security Engineering (FMSE 2004)*, pages 110–122, October 2004.
- [32] B. Watson. Beyond identity: Addressing problems that persist in an electronic mail system with reliable sender identification. In *CEAS 2004: First Conference on Email and Anti-Spam*, July 2004.

## Heuristics for enforcing security constraints

Flemming Nielson      Hanne Riis Nielson

*Informatics and Mathematical Modelling, Richard Petersens Plads bldg. 321,  
Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark*

`{nielson|riis}@imm.dtu.dk`

**Abstract.** The flow logic approach to static analysis amounts to axiomatising the admissibility of solutions to analysis problems; when axiomatised using formulae in stratified alternation-free least fixed point logic one may use efficient algorithms for computing the least admissible solutions. We extend this scenario to validate the fulfilment of additional constraints on admissible solutions; the modified development produces a least solution together with a boolean value indicating whether or not the constraints are validated or violated. — Our main contribution is the development of a deterministic heuristics for obtaining a solution that is close to the least solution while enforcing the validation of the security constraints. We illustrate it on the Bell-LaPadula mandatory access control policy where the heuristics is used to suggest modifications to the security annotations of entities in order for the security policy to hold.

### 1 Introduction

The goals of the paper are perhaps best explained by means of an analogy. In the world of type systems one frequently distinguishes between soft typing and strong typing. In *soft typing* all programs can be typed (possibly with an ever encompassing top type) and the goal is to use types to provide as much meaningful information about subprograms as possible. In *strong typing* the whole point of the type system is to reject certain programs as being ill-formed (including those that might lead to certain kinds of errors when executed) whereas providing meaningful information about subprograms is an important secondary aim. Indeed, the slogan of strong typing is that “well typed programs cannot go wrong” [8]. We might say that soft typing focuses on *solving* a type inference problem whereas strong typing focuses on *enforcing* the solvability of a type inference problem (which admittedly involves a solving phase as well) subject to additional constraints.

In this paper we consider the world of static analysis as embodied in data flow and control flow analysis. Here the view traditionally is that of *solving* an analysis problem in order to provide information that may be useful e.g. in case of a compiler generating better than naive code. When viewed in the general framework of abstract interpretation [5, 9] one usually establishes a Moore Family result showing that a least solution always exists. Given a problem *cls* we shall

write  $\mathcal{S}(cls)$  for the least solution  $\rho$ . In Section 2 we slightly extend our approach [11, 10] based on formulae in stratified alternation-free least fixed point logic.

Solutions to static analysis problems are increasingly being used also for software *validation*, e.g. for *enforcing* security policies. Quite frequently it is possible to formulate such policies as sets of constraints upon the solution  $\mathcal{S}(cls)$ . Given a problem  $cls$  with embedded constraints we shall write  $\mathcal{V}(cls)$  for the least solution  $\rho$ , as computed by  $\mathcal{S}(cls)$ , together with a boolean value  $b$  indicating the truth value of the constraints. In Section 3 we develop an extension of our approach where constraints are an integral part of the logical formalism. This development is illustrated on an example showing how to enforce that programs in a functional language never attempt to perform a function call unless the value applied is indeed a function.

This paves the way for Section 4 where we consider how to deal with a problem  $cls$  that cannot be validated, i.e. a problem  $cls$  for which  $\mathcal{V}(cls) = (\rho, false)$ . Here our goal is to develop a deterministic heuristics for finding a small modification  $\varrho$  to the solution  $\rho$  such that the problem can be validated under the assumption that the behaviour of  $\varrho$  can be admitted. This idea has in part been inspired by the non-standard approach to fixpoints explored in [7] and in our case amounts to an iterative approach to recalculating solutions. The desired result of our heuristics is  $\mathcal{H}(cls) = (\rho, \varrho)$  where  $\varrho$  is the small modification deemed necessary and  $\rho$  is the resulting least solution for which the constraints can be enforced; we may write this as  $\mathcal{V}(cls @ \varrho) = (\rho, true)$  where  $cls @ \varrho$  is a syntactic mechanism used to enforce that  $\rho \supseteq \varrho$ .

The main motivating example for this development is from the world of mandatory access control policies. Here  $\varrho$  might indicate additional entities to be considered to be within the Trusted Computing Base; we present an example showing how to formulate the Bell-LaPadula mandatory access control policy [2, 6] and how to use the heuristics for suggesting modifications to the security annotations of entities in order for the security policy to hold.

## 2 Stratified ALFP and the Succinct Solver

**Syntax.** The Alternation-free fragment of Least Fixpoint Logic (ALFP) extends Horn clauses by allowing both existential and universal quantifications in preconditions, negative queries (subject to the notion of stratification), disjunctions of preconditions, and conjunctions of conclusions.

**Definition 1.** *Given a fixed countable set  $\mathcal{X}$  of variables, a finite alphabet  $\mathcal{R}$  of predicate symbols (where all arities are at least 1) we define the set of ALFP formulae (or clause sequences),  $cls$ , together with clauses,  $cl$ , and preconditions,  $pre$ , by the grammar*

$$\begin{aligned}
 pre & ::= R(x_1, \dots, x_n) \mid \neg R(x_1, \dots, x_n) \\
 & \mid pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \mid \exists x : pre \mid \forall x : pre \\
 cl & ::= R(x_1, \dots, x_n) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid \forall x : cl \mid pre \Rightarrow cl \\
 cls & ::= cl_1, \dots, cl_k
 \end{aligned}$$

where  $x \in \mathcal{X}$ ,  $R \in \mathcal{R}$  and  $k$  is at least 1.

Occurrences of  $R$  and  $\neg R$  in preconditions are called *queries* and *negative queries*, respectively, whereas the other occurrences of  $R$  are called *assertions* of the predicate  $R$ . We write **1** for the always true clause.

**Stratification.** In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we introduce a notion of stratification similar to the one in *Datalog* [4, 1]. Intuitively, stratification ensures that a negative query is not performed until the predicate queried has been fully asserted. This is important for ensuring that once a precondition evaluates to *true* it will continue doing so even after further assertions of predicates.

**Definition 2.** We say that a formula  $cls$  is stratified w.r.t. rank whenever it has the form  $cls = cl_1, \dots, cl_k$ , and the function  $rank : \mathcal{R} \rightarrow \{0, \dots, k\}$  satisfies the following properties for all  $i = 1, \dots, k$ :

1.  $rank(R) \geq i$  for every assertion  $R$  in  $cl_i$ ;
2.  $rank(R) \leq i$  for every positive query  $R$  in  $cl_i$ ; and
3.  $rank(R) < i$  for every negative query  $\neg R$  in  $cl_i$ .

We say that a formula  $cls$  is stratified if there exists a ranking function  $rank$  such that  $cls$  is stratified w.r.t.  $rank$ .

Not all formulae are stratified and a formula may be stratified w.r.t. some ranking functions but not stratified w.r.t other ranking functions. Given a formula  $cls = cl_1, \dots, cl_k$  one can construct an *optimal ranking* function  $rank$ , i.e. one that makes  $cls$  stratified w.r.t.  $rank$  if and only if  $cls$  is stratified, by setting  $rank(R) = k$  if there are no (positive or negative) queries to  $R$  in  $cls$ , otherwise setting  $rank(R) = 0$  if there are no assertions to  $R$  in  $cls$ , and setting  $rank(R) = i$  if  $cl_i$  is the rightmost clause containing an assertion to  $R$ .

**Stratifiability.** Sometimes a clause does not have the form of a formula that is stratified w.r.t. some ranking function  $rank$  although it can easily be rearranged into such a formula. This is possible if for each subclause  $\dots R \dots \Rightarrow \dots S \dots$  we have that  $rank(R) \leq rank(S)$  and furthermore, if for each subclause  $\dots \neg R \dots \Rightarrow \dots S \dots$  we have that  $rank(R) < rank(S)$ . We shall say that a clause  $cl$  is *stratifiable* w.r.t  $rank$  whenever these conditions are met. One approach to obtaining a stratified formula  $cls$  from the clause  $cl$  is simply to construct it as  $cls = cl_1, \dots, cl_k$  where each  $cl_i$  is obtained from  $cl$  by replacing assertions of rank different from  $i$  with the clause **1**.

In a similar vein a clause  $cl$  is *stratifiable* if it is possible to construct a ranking function  $rank$  such that the clause is stratifiable w.r.t.  $rank$ . An easy test for this condition is to build a graph with predicate symbols as nodes (called  $R$  and  $S$  below) and two kinds of edges; there is a normal edge from  $R$  to  $S$  if the clause contains a subclause  $\dots R \dots \Rightarrow \dots S \dots$  and there is a fat edge from  $R$  to  $S$  if the clause contains a subclause  $\dots \neg R \dots \Rightarrow \dots S \dots$ . Then the clause is stratifiable if and only if there is no loop containing a fat edge.

Stratifiable clauses are accepted by a preprocessor to the Succinct Solver [10] and are turned into appropriately stratified formulae which are then solved.

$(\rho, \sigma) \models R(x_1, \dots, x_n)$	iff	$(\sigma(x_1), \dots, \sigma(x_n)) \in \rho(R)$
$(\rho, \sigma) \models \neg R(x_1, \dots, x_n)$	iff	$(\sigma(x_1), \dots, \sigma(x_n)) \notin \rho(R)$
$(\rho, \sigma) \models pre_1 \wedge pre_2$	iff	$(\rho, \sigma) \models pre_1$ and $(\rho, \sigma) \models pre_2$
$(\rho, \sigma) \models pre_1 \vee pre_2$	iff	$(\rho, \sigma) \models pre_1$ or $(\rho, \sigma) \models pre_2$
$(\rho, \sigma) \models \exists x : pre$	iff	$(\rho, \sigma[x \mapsto a]) \models pre$ for some $a \in \mathcal{U}$
$(\rho, \sigma) \models \forall x : pre$	iff	$(\rho, \sigma[x \mapsto a]) \models pre$ for all $a \in \mathcal{U}$
$(\rho, \sigma) \models R(x_1, \dots, x_n)$	iff	$(\sigma(x_1), \dots, \sigma(x_n)) \in \rho(R)$
$(\rho, \sigma) \models \mathbf{1}$	iff	true
$(\rho, \sigma) \models cl_1 \wedge cl_2$	iff	$(\rho, \sigma) \models cl_1$ and $(\rho, \sigma) \models cl_2$
$(\rho, \sigma) \models \forall x : cl$	iff	$(\rho, \sigma[x \mapsto a]) \models cl$ for all $a \in \mathcal{U}$
$(\rho, \sigma) \models pre \Rightarrow cl$	iff	$(\rho, \sigma) \models cl$ whenever $(\rho, \sigma) \models pre$
$(\rho, \sigma) \models cl_1, \dots, cl_k$	iff	$(\rho, \sigma) \models cl_1$ and $\dots$ and $(\rho, \sigma) \models cl_k$

**Table 1.** Semantics of preconditions, clauses and formulae.

**Constraint Satisfaction.** We take a pure approach where the logic is interpreted over a universe  $\mathcal{U}$  of constants. Given interpretations  $\rho$  and  $\sigma$  for predicate symbols and variables, respectively, we define the satisfaction relations for preconditions (denoted  $(\rho, \sigma) \models pre$ ), for clauses (denoted  $(\rho, \sigma) \models cl$ ), and for formulas (denoted  $(\rho, \sigma) \models cls$ ) as in Table 1. In particular, we write  $\rho(R)$  for the set of  $n$ -tuples  $(a_1, \dots, a_n)$  from  $\mathcal{U}^n$  associated with the  $n$ -ary predicate  $R$  and  $\sigma(x)$  for the element of  $\mathcal{U}$  denoted by the variable  $x$ .

We shall mainly be interested in *closed* formulae  $cls$ , i.e. clause sequences that have no free variables. Hence the choice of the interpretation  $\sigma$  is immaterial, so we can fix an arbitrary interpretation  $\sigma_0$ . We then call an interpretation  $\rho$  of the predicate symbols, a *solution* to the formula  $cls$  provided  $(\rho, \sigma_0) \models cls$ .

Let  $\Delta$  be the set of interpretations  $\rho$  of predicate symbols in  $\mathcal{R}$  over  $\mathcal{U}$  and let  $rank$  be a fixed ranking function.

**Definition 3.** The lexicographical ordering  $\sqsubseteq$  is defined by  $\rho_1 \sqsubseteq \rho_2$  if and only if there is some  $j \in \{0, \dots, k\}$  such that the following properties hold:

- $\rho_1(R) = \rho_2(R)$  for all  $R \in \mathcal{R}$  with  $rank(R) < j$
- $\rho_1(R) \subseteq \rho_2(R)$  for all  $R \in \mathcal{R}$  with  $rank(R) = j$
- either  $j$  is maximal in rank or  $\rho_1(R) \subset \rho_2(R)$  for at least one  $R \in \mathcal{R}$  with  $rank(R) = j$

The subset-ordering  $\subseteq$  is given by  $\rho_1 \subseteq \rho_2$  whenever  $\forall R \in \mathcal{R} : \rho_1(R) \subseteq \rho_2(R)$ .

**Fact 1.** If  $\rho_1 \subseteq \rho_2$  then  $\rho_1 \sqsubseteq \rho_2$  (but not necessarily vice versa).

*Proof.* Let  $j' \in \{-1, \dots, k\}$  be maximal such that  $rank(R) \leq j' \implies \rho_1(R) = \rho_2(R)$  and let  $j$  be the smaller of  $j' + 1$  and  $k$ ; then it is immediate to show that  $\rho_1 \sqsubseteq \rho_2$  using that  $\rho_1 \subseteq \rho_2$ . (Conversely, consider  $\rho_1(R_1) = \rho_2(R_2) = \emptyset$  and  $\rho_1(R_2) = \rho_2(R_1) = \{\cdot\}$  where  $rank(R_i) = i$ ; then  $\rho_1 \sqsubseteq \rho_2$  holds (take  $j = 1$ ) but  $\rho_1 \subseteq \rho_2$  fails.)  $\square$

**Proposition 1 (from [10]).** *The set  $\Delta = (\Delta, \sqsubseteq)$  forms a complete lattice.*

*The solution set  $\Delta_{cls} = \{\rho \in \Delta \mid (\rho, \sigma_0) \models cls\}$  forms a Moore family, i.e. it is closed under greatest lower bounds (w.r.t.  $\sqsubseteq$ ), whenever  $cls$  is a closed and stratified formula.*

**The Succinct Solver.** In the sequel we shall only be interested in the least solution  $\rho$  as guaranteed by the above proposition; formally it is given by

$$\mathcal{S}(cls) = \sqcap \{\rho \in \Delta \mid (\rho, \sigma_0) \models cls\}$$

and is the solution computed by the Succinct Solver [10].

For the purposes of this paper it suffices with the following imprecise account of the operation of the Succinct Solver; the actual algorithm [10] operates in a considerably more intelligent manner. Given a stratified clause  $cls$  we may construct two functionals  $N_{cls}$  and  $F_{cls}$ . For this we shall write  $\rho = \rho_1 \cup \dots \cup \rho_k$  where each  $\rho_i$  defines the predicates of rank  $i$  (where we assume for simplicity of presentation that all predicates have non-zero rank). We set  $N_{cls}(\rho_1 \cup \dots \cup \rho_k) = (\varrho_1 \cup \dots \cup \varrho_k)$  whenever  $\varrho_1 \cup \dots \cup \varrho_k$  constitutes the *new* contribution to the predicates arising from *one pass* through  $cls$ . Then we set

$$F_{cls}(\rho) = \begin{cases} \rho & \text{if } N_{cls}(\rho) = (\perp, \dots, \perp) \\ \rho \cup \varrho_i & \text{if } N_{cls}(\rho) = (\perp, \dots, \varrho_i, \dots) \text{ and } \varrho_i \neq \perp \end{cases}$$

where the intention is that  $i$  indicates the first component of  $N_{cls}(\rho)$  that is not  $\perp$ . The Succinct Solver may then be described as operating until stabilisation of  $F_{cls}$ , i.e.  $\mathcal{S}(cls) = \sqcup_i F_{cls}^i(\perp, \dots, \perp)$ .

For a partial order  $\leq$  (e.g.  $\sqsubseteq$  or  $\subseteq$ ) we shall say that the functional  $F$  is  $\leq$ -monotonic if  $\forall \rho_1, \rho_2 : \rho_1 \leq \rho_2 \implies F(\rho_1) \leq F(\rho_2)$  and  $\leq$ -extensive if  $\forall \rho : \rho \leq F(\rho)$ . We then have:

**Fact 2.** *The functional  $F_{cls}$  is  $\subseteq$ -extensive and  $\sqsubseteq$ -extensive. (The functional need not be  $\subseteq$ -monotonic nor  $\sqsubseteq$ -monotonic.)*

*Proof.* That  $F_{cls}$  is  $\subseteq$ -extensive is obvious by construction; that  $F_{cls}$  is  $\sqsubseteq$ -extensive follows from Fact 1. (To see that  $F_{cls}$  need not be  $\sqsubseteq$ -monotonic consider the scenario in the proof of Fact 1 and take  $cls = R_1(\cdot)$ . To see that  $F_{cls}$  need not be  $\subseteq$ -monotonic take  $\rho_i(R_j) = \emptyset$  except  $\rho_2(R_1) = \{\cdot\}$  and take  $cls = \neg R_1(\cdot) \Rightarrow R_2(\cdot)$ .)  $\square$

### 3 Constraints on ALFP

*Example 1.* As a motivating example consider a simple functional language

$$e^l ::= \text{tt}^l \mid \text{ff}^l \mid x^l \mid (\lambda x. e_0^{l_0})^l \mid (e_1^{l_1} e_2^{l_2})^l \mid (\text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2})^l$$

where  $e^l$  ranges over labelled expressions. A control flow analysis keeps track of which values (truth values and lambda abstractions) reach which points in the



program. We axiomatise it using these predicates:

- $C(l, v)$  indicates that the set of values arising at a subexpression labelled  $l$  may contain the value  $v$ ,
- $R(x, v)$  indicates that in the environment the variable  $x$  may be bound to the value  $v$ ,
- $P(l, v)$  indicates that the value  $v$  may be an actual parameter to a  $\lambda$ -abstraction whose body is labelled  $l$ ,
- $B(v)$  indicates that  $v$  is a boolean value in the program,
- $A(l)$  indicates that  $l$  labels the body of a  $\lambda$ -abstraction in the program.

For each subprogram of a given program we then generate clauses as follows:

$$\begin{aligned}
 \text{tt}^l &\mapsto C(l, \text{tt}) \wedge B(\text{tt}) \\
 \text{ff}^l &\mapsto C(l, \text{ff}) \wedge B(\text{ff}) \\
 x^l &\mapsto \forall v : R(x, v) \Rightarrow C(l, v) \\
 (\lambda x. e_0^{l_0})^l &\mapsto C(l, l_0) \wedge A(l_0) \wedge \forall v : P(l_0, v) \Rightarrow R(x, v) \\
 (e_1^{l_1} e_2^{l_2})^l &\mapsto \forall u : C(l_1, u) \Rightarrow ((\forall v : C(l_2, v) \Rightarrow P(u, v)) \wedge \\
 &\quad (\forall w : C(u, w) \Rightarrow C(l, w))) \\
 (\text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2})^l &\mapsto \forall v : (C(l_1, v) \vee C(l_2, v)) \Rightarrow C(l, v)
 \end{aligned}$$

Here the variables (like  $u$ ,  $v$  and  $w$ ) range over the universe  $\mathcal{U}$  that consists of the basic values  $\text{tt}$  and  $\text{ff}$  and all labels. In the clause for an application  $(e_1^{l_1} e_2^{l_2})^l$  a typical value of  $u$  will be some label  $l_0$  denoting a  $\lambda$ -abstraction. The overall clause generated for the entire program is the conjunction of all the clauses above. It is clearly a stratifiable clause w.r.t. a rank function given by  $\text{rank}(B) = \text{rank}(A) = 1$  and  $\text{rank}(C) = \text{rank}(R) = \text{rank}(P) = 2$ .

In order to *validate* the correct behaviour of the program it would be prudent to impose constraints ensuring that only functions are applied to arguments and that only booleans are used to discriminate between branches of conditionals. Such constraints can be checked by evaluating the following formulae on the least solution to the clause generated above:

- for an application  $(e_1^{l_1} e_2^{l_2})^l$  check that  $\forall u : C(l_1, u) \Rightarrow A(u)$ , and
- for a conditional  $(\text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2})^l$  check that  $\forall u : C(l_0, u) \Rightarrow B(u)$ .

It would be preferable if the constraints could be integrated with the specification of the clause generation. However, this is not possible because rather than giving rise to a constraint that may evaluate to *false*, it would give rise to merely adding new “spurious elements” to the predicates  $A$  and  $B$  like asserting  $A(\text{ff})$ ,  $A(\text{tt})$  or  $B(l_0)$ . In this respect it is worth pointing out that the least solution as produced by  $\mathcal{S}$  ensures that no such “spurious elements” are part of the least solution (because they are not explicitly demanded to be so by the clause constructed above).  $\square$

**Constrained ALFP.** We therefore extend the syntax of ALFP by allowing explicit occurrences of constraints. We distinguish between an assertion  $R(x_1, \dots, x_n)$  and a constraint by writing the latter as  $R!(x_1, \dots, x_n)$ .

	IGNORE	ENFORCE	OBSERVE
$R!(\mathbf{x})$	<b>1</b>	$R(\mathbf{x})$	$\neg R(\mathbf{x}) \Rightarrow R^E(\mathbf{x})$
$R(\mathbf{x})$	$R(\mathbf{x})$	$R(\mathbf{x})$	$R(\mathbf{x})$
<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>
$cl_1 \wedge cl_2$	$\text{IGNORE}(cl_1) \wedge \text{IGNORE}(cl_2)$	$\text{ENFORCE}(cl_1) \wedge \text{ENFORCE}(cl_2)$	$\text{OBSERVE}(cl_1) \wedge \text{OBSERVE}(cl_2)$
$\forall x : cl$	$\forall x : \text{IGNORE}(cl)$	$\forall x : \text{ENFORCE}(cl)$	$\forall x : \text{OBSERVE}(cl)$
$pre \Rightarrow cl$	$pre \Rightarrow \text{IGNORE}(cl)$	$pre \Rightarrow \text{ENFORCE}(cl)$	$pre \Rightarrow \text{OBSERVE}(cl)$
$cl_1, \dots, cl_k$	$\text{IGNORE}(cl_1), \dots, \text{IGNORE}(cl_k)$	$\text{ENFORCE}(cl_1), \dots, \text{ENFORCE}(cl_k)$	$\text{OBSERVE}(cl_1), \dots, \text{OBSERVE}(cl_k)$

**Table 2.** Ignoring, enforcing or observing the constraints in clauses and formulae.

**Definition 4.** The set of constrained ALFP clauses,  $cl$ , are given by

$$cl ::= R!(x_1, \dots, x_n) \mid R(x_1, \dots, x_n) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid \forall x : cl \mid pre \Rightarrow cl$$

whereas constrained preconditions and formulae are as in Definition 1.

We say that a constrained formula  $cls$  is stratified w.r.t.  $rank$  whenever it has the form  $cls = cl_1, \dots, cl_k$ , and the function  $rank : \mathcal{R} \rightarrow \{0, \dots, k\}$  satisfies the following properties for all  $i = 1, \dots, k$ :

1.  $rank(R) < i$  for every constraint  $R!$  in  $cl_i$ ;
2.  $rank(R) \geq i$  for every assertion  $R$  in  $cl_i$ ;
3.  $rank(R) \leq i$  for every positive query  $R$  in  $cl_i$ ; and
4.  $rank(R) < i$  for every negative predicate  $\neg R$  in  $cl_i$ .

We say that a constrained formula  $cls$  is stratified if there exists a ranking function  $rank$  such that  $cls$  is stratified w.r.t.  $rank$ .

In the definition of stratified we have taken the view (to become even clearer when discussing constraint violations below) that a predicate should not be used as a constraint until it has been fully asserted.

The *optimal ranking* function is constructed much as before:  $rank(R) = k$  if there are no (positive or negative) queries to  $R$  nor constraints on  $R$  in  $cls$ , otherwise  $rank(R) = 0$  if there are no assertions to  $R$  in  $cl$ , and  $rank(R) = i$  if  $cl_i$  is the rightmost clause containing an assertion to  $R$ .

The considerations of *stratifiability* apply mutatis mutandis; as before there is a normal edge from  $R$  to  $S$  if the clause contains a subclause  $\dots R \dots \Rightarrow \dots S \dots$  and there is a fat edge from  $R$  to  $S$  if the clause contains a subclause  $\dots \neg R \dots \Rightarrow \dots S \dots$ . (In both cases  $S$  denotes an assertion rather than a constraint.) Much as before a constrained clause  $cl$  is stratifiable if and only if there is no loop containing a fat edge. However, when constructing the stratified formula  $cls$  we may have to introduce a new rank  $k+1$  and construct it as  $cls = cl_1, \dots, cl_{k+1}$  where each  $cl_i$  is obtained from  $cl$  by replacing assertions of rank different from  $i$  with the clause **1** and furthermore replacing constraints of rank different from  $i-1$  with the clause **1**. As an example, the clause  $R(a) \Rightarrow (R(b) \wedge R!(c))$  becomes  $R(a) \Rightarrow (R(b) \wedge \mathbf{1})$ ,  $R(a) \Rightarrow (\mathbf{1} \wedge R!(c))$ .

**Constraint Validation.** We shall deal with the semantics of constrained ALFP in a syntactic manner, by defining two ways in which to translate a constrained formula into a formula of ALFP.

The function `IGNORE` simply replaces  $\dots R!(\bar{x}) \dots$  by  $\dots \mathbf{1} \dots$  and hence ignores the constraints imposed (see Table 2 for the details):

$$\text{IGNORE}(\dots R!(\mathbf{x}) \dots) = \dots \mathbf{1} \dots$$

It is useful for extracting the constraint-free part of the formula for which the least solution is desired. If  $cls$  is a stratified and constrained formula then clearly  $\text{IGNORE}(cls)$  is a stratified formula of ALFP.

Similarly, the function `ENFORCE` replaces  $\dots R!(\bar{x}) \dots$  by  $\dots R(\bar{x}) \dots$  and hence ignores the distinction between constraints and assertions (see Table 2 for the details):

$$\text{ENFORCE}(\dots R!(\mathbf{x}) \dots) = \dots R(\mathbf{x}) \dots$$

It is useful for extracting a formula that can be used to check whether or not the constraints are fulfilled. However, even if  $cls$  is a stratified and constrained formula, the formula  $\text{ENFORCE}(cls)$  need not be a stratified formula of ALFP; as an example consider  $\mathbf{1}, \neg R(a) \Rightarrow R!(b)$  where  $R$  has rank 1.

Given a closed, stratified and constrained formula  $cls$ , we may define the *validation* function  $\mathcal{V}(cls) = (\rho, b)$  as follows:

$$\begin{aligned} \mathcal{V}(cls) = & \begin{array}{l} \text{let } \rho = \mathcal{S}(\text{IGNORE}(cls)) \text{ in} \\ \text{let } b = ((\rho, \sigma_0) \models \text{ENFORCE}(cls)) \text{ in} \\ (\rho, b) \end{array} \end{aligned}$$

Here  $\mathcal{V}(cls) = (\rho, b)$  means that  $\rho$  is the least solution when ignoring the constraints and  $b$  indicates whether or not the constraints are validated.

*Example 2.* Returning to Example 1 we can now write the clauses to be generated for application and conditionals as follows:

$$\begin{aligned} (e_1^{l_1} e_2^{l_2})^l & \mapsto \forall u : C(l_1, u) \Rightarrow (A!(u) \wedge \\ & \quad (\forall v : C(l_2, v) \Rightarrow P(u, v)) \wedge \\ & \quad (\forall w : C(u, w) \Rightarrow C(l, w))) \\ (\text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2})^l & \mapsto \forall u : C(l_0, u) \Rightarrow B!(u) \quad \wedge \\ & \quad \forall v : (C(l_1, v) \vee C(l_2, v)) \Rightarrow C(l, v) \end{aligned}$$

These clauses are stratifiable w.r.t. the rank function of Example 1. Furthermore, `IGNORE` translates the clauses generated into clauses that are logically equivalent to the ones of Example 1, whereas `ENFORCE` translates the clauses generated into clauses that are logically equivalent to the conjunction of the ones of Example 1 together with the constraints imposed there.  $\square$

**Constraint Violation.** An alternative approach to calculating  $\mathcal{V}(cls) = (\rho, b)$  where  $b$  indicates whether or not the least solution  $\rho$  validates the constraints is to directly record the violations to the constraints, if any. This corresponds to making use of so-called observation predicates [3] for tracking the violations to constraints. Intuitively we should be able to show that the constraints are validated if and only if there are no violations; this will be the result of Proposition 2 below.

To describe the alternative approach we define a function **OBSERVE** that translates  $\dots R!(\bar{x}) \dots$  to  $\dots (\neg R(\bar{x}) \Rightarrow R^E(\bar{x})) \dots$  where we assume that to each “ordinary” predicate  $R$  there potentially is an “observation” predicate  $R^E$  (see Table 2 for the details):

$$\text{OBSERVE}(\dots R!(\mathbf{x}) \dots) = \dots \neg R(\mathbf{x}) \Rightarrow R^E(\mathbf{x}) \dots$$

We shall write  $\mathcal{R}^\circ$  for the set of ordinary predicates and  $\mathcal{R}^E$  for the set of observation predicates and assume that  $\mathcal{R}$  is the disjoint union of these two sets. Furthermore, we extend the given ranking function  $rank$  by setting  $rank(R^E) = k + 1$  for all observation predicates  $R^E$  (regardless of the rank of  $R$ ).

We can now explain the use of observation predicates as an alternative strategy for defining the validation function  $\mathcal{V}$ :

**Proposition 2.**  $\mathcal{V}(cls) = (\rho, b)$  holds if and only if

$$\rho = \mathcal{S}(\text{OBSERVE}(cls)) \upharpoonright_{\mathcal{R}^\circ}$$

$$b = \bigwedge_{R^E \in \mathcal{R}^E} (\rho(R^E) = \emptyset)$$

where  $\dots \upharpoonright_{\mathcal{R}^\circ}$  denotes the restriction to ordinary predicates only.

## 4 Heuristics for Success

*Example 3.* As a fairly substantial motivating example consider a simplified presentation of the Bell-LaPadula mandatory access control policy for enforcing confidentiality [2, 6]. The basic entities are subjects (e.g. programs or users), objects (e.g. files), operations (read and write) and security levels (high and low).

The actual operations are specified by statements of the form **read**( $s, o$ ) for indicating that the subject  $s$  is initiating a read-operation on the object  $o$  and similarly **write**( $s, o$ ) for indicating that the subject  $s$  is initiating a write-operation on the object  $o$ .

The discretionary part of the access control policy is syntactically specified by statements of the form **readable**( $o : s_1, \dots, s_n$ ) for indicating that the object  $o$  may be read by any one of the subjects  $s_1, \dots, s_n$  and by **writable**( $o : s_1, \dots, s_n$ ) for indicating that the object  $o$  may be written by any one of the subjects  $s_1, \dots, s_n$ .

The mandatory part of the access control policy is syntactically specified by statements of the form **subject**( $s : \phi$ ) for indicating that the subject  $s$  is

allowed to operate at security level  $\phi$  (being one of  $H$  or  $L$ ) and by  $\text{object}(o : \phi)$  for indicating that the object  $o$  may be accessed at security level  $\phi$  (being one of  $H$  or  $L$ ).

For the purposes of specifying the security policy we shall view the semantics as operating over configurations of the form  $(S, O, M, B)$ . Here  $S(s, \phi)$  records that the subject  $s$  has been previously allowed to operate at security level  $\phi$ ; in the classical presentation [6] it aims at capturing  $f_C(s) = f_S(s) = \phi$ . Similarly,  $O(s, \phi)$  records that the object  $o$  has been previously allowed to be manipulated at security level  $\phi$ ; in the classical presentation [6] it aims at capturing  $f_O(o) = \phi$ . Furthermore,  $M(s, o, r)$  captures that the object  $o$  has previously been recorded as readable by subject  $s$  and  $M(s, o, w)$  captures that the object  $o$  has previously been recorded as writable by subject  $s$ ; this is as in the classical presentation [6]. Finally,  $B(s, o, r)$  indicates that in the current state the subject  $s$  has initiated reading the object  $o$  and  $B(s, o, w)$  indicates that in the current state the subject  $s$  has initiated writing the object  $o$ ; also this is as in [6].

We shall develop a simple flow-insensitive analysis for keeping track of these operations and for enforcing the Bell-LaPadula mandatory access control policy (called **mac** and formally defined below). Since it is flow-insensitive it may operate over an “abstract state”  $(S, O, M, B)$  as explained above. For the various operations we generate constraints as follows:

$$\begin{aligned} \text{read}(s, o) &\mapsto B(s, o, r) \wedge \text{mac} \\ \text{write}(s, o) &\mapsto B(s, o, w) \wedge \text{mac} \\ \text{readable}(o : s_1, \dots, s_n) &\mapsto M(o, s_1, r) \wedge \dots \wedge M(o, s_n, r) \\ \text{writable}(o : s_1, \dots, s_n) &\mapsto M(o, s_1, w) \wedge \dots \wedge M(o, s_n, w) \\ \text{subject}(s : \phi) &\mapsto S(s, \phi) \\ \text{object}(o : \phi) &\mapsto O(o, \phi) \end{aligned}$$

The clause for the Bell-LaPadula mandatory access control policy is:

$$\text{mac} = \left( \begin{array}{l} \forall s, o, o' : B(s, o, w) \wedge B(s, o', r) \\ \Rightarrow M!(s, o, w) \wedge M!(s, o', r) \quad \wedge \\ (S(s, H) \wedge \neg S(s, L)) \Rightarrow O!(o, H) \quad \wedge \\ (O(o, L) \wedge \neg O(o, H)) \Rightarrow (S!(s, L) \wedge O!(o', L)) \quad \wedge \\ (O(o', H) \wedge \neg O(o', L)) \Rightarrow (O!(o, H) \wedge S!(s, H)) \quad \wedge \\ (S(s, L) \wedge \neg S(s, H)) \Rightarrow O!(o', L) \end{array} \right)$$

Here the first line considers a situation where a subject  $s$  is simultaneously writing an object  $o$  and reading an object  $o'$ . The second line enforces that these operations have indeed been previously allowed as indicated by the access control matrix  $M$ . The remaining lines enforce that the security classification of  $o$  dominates those of  $s, o'$  and that the security classification of  $o'$  is dominated by those of  $s, o$  (relying once more on  $f_C = f_S$  in the classical presentation of [6]).

The clause generated is clearly stratifiable. A simple choice of a ranking function is to take  $\text{rank}(S) = 1$ ,  $\text{rank}(O) = 1$ ,  $\text{rank}(M) = 1$  and  $\text{rank}(B) = 1$ .

A more interesting choice (as we shall argue shortly) is to take  $rank(S) = 1$ ,  $rank(O) = 2$ ,  $rank(M) = 3$  and  $rank(B) = 4$ .

To be a bit more concrete consider a program involving one subject `sub` and two objects `ob1` and `ob2`:

```
subject(sub:H);
object(ob1:L); readable(ob1:sub); writable(ob1:sub);
object(ob2:L); readable(ob2:sub); writable(ob2:sub);
read(sub,ob2); write(sub,ob1);
```

Here there is a violation of the mandatory part of the access control policy: when `sub` reads `ob2` and writes `ob1` the security level of `sub` (which is H) must be dominated by that of `ob1` (which is L).

Assuming that the program is intended to be legitimate we must modify the security annotations such that `mac` holds. Intuitively, there are two ways to do so: one is to downgrade `sub` to L, the other is to upgrade `ob1` to H. From a security policy point of view it is usually preferred to upgrade the objects rather than downgrading the subjects (see [6] for a discussion). In the present case this means that we prefer to add “spurious elements” to relations like  $O$  rather than relations like  $S$ .

Hence we would like a general heuristics that, based on the rank-information automatically suggests remedial actions. We shall decide to go for an approach where we prefer to remedy the values of higher-rank relations rather than lower-rank relations; in operational terms this means restricting how far the Succinct Solver needs to backtrack and corresponds to its overall mode of operation as described in Section 2. In the present case this suggests taking  $rank(S) = 1$ ,  $rank(O) = 2$ ,  $rank(M) = 3$  and  $rank(B) = 4$ .  $\square$

**Acceptable heuristics.** So far we have been content with an optimal algorithm  $\mathcal{S}$  for solving an analysis problem expressed by a closed and stratified formula, and an optimal algorithm  $\mathcal{V}$  for solving and validating the constraints as expressed by a closed and stratified constrained formula.

Turning to the construction of a heuristic algorithm  $\mathcal{H}$  we shall shortly formulate a notion of optimality and show that in general there does not exist an optimal algorithm. Hence we shall consider candidate functions  $\mathcal{H}$  of the form  $\mathcal{H}(cls) = (\rho, \varrho)$  and define when we consider them to be *acceptable*. Henceforth, we shall write  $cls \in \mathcal{F}[rank]$  to express that  $cls$  is a closed constrained formula that is stratified w.r.t.  $rank$ .

The first part of the development amounts to allowing  $\varrho$  to be freely chosen but to demand that  $\rho$  is constructed in an optimal manner from  $cls$  and  $\varrho$  and to show that this is always possible.

**Definition 5.** A function of the form  $\mathcal{H}(cls) = (\rho, \varrho)$  is a heuristics provided that  $\rho$  is least such that  $\rho \supseteq \varrho$  and  $(\rho, \sigma_0) \models \text{IGNORE}(cls)$  whenever  $cls \in \mathcal{F}[rank]$ .

**Proposition 3.** If  $cls \in \mathcal{F}[rank]$  and  $\varrho$  is given, then there always exists a least  $\rho$  such that  $\rho \supseteq \varrho$  and  $(\rho, \sigma_0) \models \text{IGNORE}(cls)$ .

*Proof.* The proof amounts to showing that

$$\rho = \sqcap \{ \rho' \in \Delta \mid (\rho', \sigma_0) \models \text{IGNORE}(cls) \wedge \rho' \supseteq \varrho \}$$

always exists and fulfils the demands. Many strategies of proof can be used, but for the purposes of this presentation we restrict ourselves to the case  $\forall R \in \mathcal{R} : \text{rank}(R) > 0$  where we can give a simple “syntactic” proof. Given  $cls = cl_1, \dots, cl_k$  we define the formula  $cls@_{\varrho} = cl'_1, \dots, cl'_k$  by setting

$$cl'_i = cl_i \wedge \bigwedge_{a \in \varrho(R), \text{rank}(R)=i} R(a)$$

Clearly  $(\rho, \sigma_0) \models \text{IGNORE}(cls)@_{\varrho}$  is equivalent to  $(\rho, \sigma_0) \models \text{IGNORE}(cls) \wedge \rho \supseteq \varrho$  and hence the above formula for  $\rho$  amounts to  $\rho = \mathcal{S}(\text{IGNORE}(cls)@_{\varrho})$ .  $\square$

The second part of the development amounts to ensuring that  $\varrho$  contains all the “spurious elements” that need to be admitted in order to fulfil the constraints.

**Definition 6.** A heuristics in the sense of Definition 5 of the form  $\mathcal{H}(cls) = (\rho, \varrho)$  is acceptable provided that  $(\rho, \sigma_0) \models \text{ENFORCE}(cls)$  whenever  $cls \in \mathcal{F}[\text{rank}]$ .

**Fact 3.** An acceptable heuristics exists.

*Proof.* Take  $\mathcal{H}(cls) = (\top, \top)$ .  $\square$

To be able to choose between acceptable heuristics we shall define a partial order for comparing them. We base it on the lexicographic order (rather than the subset-order) in order to capture the intentions expressed towards the end of Example 3.

**Definition 7.** A heuristics  $\mathcal{H}_1$  is better than a heuristics  $\mathcal{H}_2$ , and equivalently  $\mathcal{H}_2$  is worse than  $\mathcal{H}_1$ , provided that for all  $cls \in \mathcal{F}[\text{rank}]$ : if  $\mathcal{H}_1(cls) = (\rho_1, \varrho_1)$  and  $\mathcal{H}_2(cls) = (\rho_2, \varrho_2)$  then  $\varrho_1 \sqsubseteq \varrho_2$ .

We prefer this definition to the alternative where we instead compare the resulting solutions, as in  $\rho_1 \sqsubseteq \rho_2$ , because of its focus on the “spurious elements” that need to be added. Clearly the heuristics indicated in the proof of Fact 3 is worse than all others.

It would be natural to try to find the best acceptable heuristics. Unfortunately, this is not possible, i.e. we do not have the analogue of a Moore Family result for acceptable heuristics.

**Proposition 4.** There exists no best acceptable heuristics.

*Proof.* It suffices to find a stratified constrained formula  $cls$  for which no acceptable heuristics  $\mathcal{H}$  can give a best result. For this consider the formula

$$1, \quad ((\neg R(a) \wedge \neg R(b)) \Rightarrow R!(a)) \wedge ((\neg R(a) \wedge \neg R(b)) \Rightarrow R!(b))$$

where  $\text{rank}(R) = 1$  and the universe is  $\mathcal{U} = \{a, b\}$ . A heuristics  $\mathcal{H}$  must produce one of the following pairs  $(\rho_i, \varrho_i)$ :

1.  $\varrho_1(R) = \emptyset$  and  $\rho_1(R) = \emptyset$ ;
2.  $\varrho_2(R) = \{a\}$  and  $\rho_2(R) = \{a\}$ ;
3.  $\varrho_3(R) = \{b\}$  and  $\rho_3(R) = \{b\}$ ;
4.  $\varrho_4(R) = \{a, b\}$  and  $\rho_4(R) = \{a, b\}$ .

Of these 2–4 are acceptable and 2–3 are acceptable and minimal. Since there are two incompatible minimal choices no optimal choice of an acceptable heuristics can exist.  $\square$

**A good acceptable heuristics.** We now consider a class of parameterised iterative heuristic algorithms  $\mathcal{H}[\text{choose}, \text{take}]$ . Here **choose** is a function intended to select an index from a set of indices, and **take** is a function intended to select part of a partial solution; it will turn out that our preferred candidate has  $\text{choose} = \text{max}$  and  $\text{take} = \lambda \varrho. \varrho$  (i.e. the identity).

The definition is given in Table 3. It accepts as input a closed, constrained and stratified formula  $cls$  w.r.t. a ranking function  $rank$  (that without loss of generality is assumed to use non-zero ranks only) and produces the pair  $(\rho, \varrho)$ . It operates in an iterative manner, “backpropagating” any violations to constraints. We use the function  $\text{OBSERVE}(\dots R!(\mathbf{x}) \dots) = \dots \neg R(\mathbf{x}) \Rightarrow R^E(\mathbf{x}) \dots$  of Table 2 and we write  $\mathcal{R}_i^\circ$  for the set of ordinary predicates of rank  $i$  and similarly  $\mathcal{R}_j^E$  for the set of error predicates corresponding to ordinary predicates of rank  $j$  and finally we use  $\dots |_{\mathcal{R}'}$  to denote the restriction to a set  $\mathcal{R}'$  of predicates.

We shall consider two algorithms:  $\mathcal{H}[\text{max}, \lambda \varrho. \varrho]$  that selects the maximum index for which a violation of the constraints have been observed and then selects the entire error-component corresponding to this index; and  $\mathcal{H}[\text{min}, \lambda \varrho. \varrho]$  that selects the minimum index for which a violation of the constraints have been observed and then selects the entire error-component corresponding to this index.

The correct operation of the algorithm is guaranteed by:

**Proposition 5.**  $\mathcal{H}[\text{choose}, \text{take}]$  is an acceptable heuristics if  $\forall I : \text{choose}(I) \in I$  and  $\forall \tilde{\rho}_i : \tilde{\rho}_i \neq \perp \Rightarrow \text{take}(\tilde{\rho}_i) \neq \perp$ .

*Proof.* The assumptions suffice for proving that  $\mathcal{H}[\text{choose}, \text{take}]$  always terminates because in each iteration of the loop the tuple  $((\varrho_1, \dots, \varrho_i, \perp, \dots, \perp), i)$  will be strictly increasing w.r.t. the lexicographic order defined using  $\sqsubseteq$  for  $(\varrho_1, \dots, \varrho_i, \perp, \dots, \perp)$  and  $\leq$  for  $i$ .  $\square$

*Example 4.* Returning to Example 3,  $\mathcal{H}[\text{max}, \lambda \varrho. \varrho]$  suggests the remedial action of upgrading **ob1** to **H** by producing  $\varrho(O) = \{(\text{ob1}, \text{H})\}$ . This is preferable to  $\mathcal{H}[\text{min}, \lambda \varrho. \varrho]$  that suggests the remedial action of downgrading **sub** to **L** by producing  $\varrho(S) = \{(\text{sub}, \text{L})\}$ .  $\square$

While it is easy to find examples (like the one above) where  $\mathcal{H}[\text{max}, \lambda \varrho. \varrho]$  is indeed better than  $\mathcal{H}[\text{min}, \lambda \varrho. \varrho]$ , we cannot state this in general. For an example of a formula where  $\mathcal{H}[\text{min}, \lambda \varrho. \varrho]$  performs better than  $\mathcal{H}[\text{max}, \lambda \varrho. \varrho]$  consider

$$R(a), \quad S(a), \quad T(a), \quad (\neg S(b) \wedge \neg T(b)) \Rightarrow (S!(b) \wedge T!(b)), \quad T(b) \Rightarrow R!(b)$$



```

INPUT  $cls = cl_1, \dots, cl_k$  and  $rank$ 
    s.t.  $cls \in \mathcal{F}[rank]$ 
    s.t.  $rank$  uses non-zero ranks only
OUTPUT  $(\rho, \varrho) = (\rho_1 \cup \dots \cup \rho_k), (\varrho_1 \cup \dots \cup \varrho_k)$ 
METHOD  $i := 1; \varrho_i := \perp;$ 
    while  $i \leq k$  do
         $\tilde{\rho} := \mathcal{S}(\text{OBSERVE}((cl_1, \dots, cl_i) @ (\varrho_1 \cup \dots \cup \varrho_i)));$ 
         $b := \bigwedge_{R^E \in \mathcal{R}^E} (\tilde{\rho}(R^E) = \emptyset);$ 
        if  $b$  then  $\rho_1 := (\tilde{\rho} \upharpoonright_{\mathcal{R}_1^E}); \dots; \rho_i := (\tilde{\rho} \upharpoonright_{\mathcal{R}_i^E}); i := i + 1; \varrho_i := \perp;$ 
        else  $i := \text{choose}\{j \mid \exists R^E \in \mathcal{R}_j^E : \tilde{\rho}(R^E) \neq \emptyset\}; \varrho_i := \varrho_i \cup \text{take}(\tilde{\rho} \upharpoonright_{\mathcal{R}_i^E});$ 

```

**Table 3.** The family of acceptable heuristics  $\mathcal{H}[\text{choose}, \text{take}]$ .

where  $rank(R) = 1$ ,  $rank(S) = 2$  and  $rank(T) = 3$ . Here  $\mathcal{H}[\text{min}, \lambda \varrho. \varrho]$  produces  $\varrho(R) = \emptyset, \varrho(S) = \{b\}, \varrho(T) = \emptyset$  and  $\rho(R) = \{a\}, \rho(S) = \{a, b\}, \rho(T) = \{a\}$ , whereas  $\mathcal{H}[\text{max}, \lambda \varrho. \varrho]$  produces  $\varrho(R) = \{b\}, \varrho(S) = \emptyset, \varrho(T) = \{b\}$  and  $\rho(R) = \{a, b\}, \rho(S) = \{a\}, \rho(T) = \{a, b\}$ .

## 5 Conclusion

We have extended the flow logic approach to static analysis: instead of merely specifying admissible solutions to analysis problems we additionally specify constraints to be enforced on the admissible solutions. Our use of a simple syntactic distinction between assertions,  $R(\mathbf{x})$ , and constraints,  $R!(\mathbf{x})$ , have resulted in very readable specifications, as was illustrated on a simple “typing example” for the  $\lambda$ -calculus.

Our main contribution is the development of a heuristics that facilitates loosening some of the constraints in order for a security policy to hold for selected programs. The motivating example for this development has been the Bell-LaPadula mandatory access control policy. Our strategy for loosening constraints has been to keep the logical formulae unchanged but to admit more elements in the constraining predicates; we have generally referred to these as “spurious elements”. We have studied and proposed heuristics for iteratively recomputing least solutions to analysis problems in such a way that the final solution adheres to the constraints posed. Also we have shown that a heuristics is all that can be hoped for.

It is worth pointing out that this inherently iterative procedure can still be formulated in a logical setting; this distinguishes our approach from that of others (which to our knowledge remains unpublished). The general mechanism facilitating our development has been to use the rank-information to express the order of preference for adding the “spurious elements”. We believe there to be a fair amount of flexibility in the choice of a ranking function  $rank$  for turning a stratifiable clause into a formula that is stratified wrt.  $rank$ ; generally it should be possible to assign low ranks to predicates recording simple observations from

the program whereas the predicates carrying the actual control flow information may be so interdependent that they all need to get the same rank.

Further work is needed for determining the extent to which this approach is applicable to other security features; possibilities include restricting the behaviour of subjects inside the Trusted Computing Base and identifying the need for enlarging the Trusted Computing Base.

*Acknowledgements.* This work has been supported by the Danish Natural Science Research Council project LoST (21-02-0507). We should like to thank René Rydhof Hansen for discussions and ideas, Hongyan Sun for working with us on some preliminary ideas and the referees for their considerations and suggestions.

## References

1. K. Apt, H. Blair, and A. Walker. A theory of declarative programming. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufman, 1988.
2. D. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report ESDTR-75-306, MTR-2547, MITRE Corporation, 1975.
3. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, H. Riis Nielson: Automatic Validation of Protocol Narration. In *Proc. of the 16th Computer Security Foundations Workshop (CSFW)*, pages 126–140, IEEE Computer Society Press, 2003.
4. A. Chandra, D. Harel: Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.
5. P. Cousot, R. Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoint. *Proc. of 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, ACM Press, 1977.
6. D. Gollmann: *Computer Security*. Wiley, 1999.
7. Z. Manna, A. Shamir: The Optimal Approach to Recursive Programs. *Communications of the ACM*, pages 824–831. ACM Press, 1977.
8. R. Milner: A Theory of Typed Polymorphism in Programming. *Journal of Computer and System Sciences*, Vol. 17, pages 348–275, 1978.
9. F. Nielson, H. Riis Nielson, C. Hankin: *Principles of Program Analysis*. Springer Verlag, 2nd printing, 2005.
10. F. Nielson, H. Seidl, and H. Riis Nielson: A Succinct Solver for ALFP. *Nordic Journal of Computing*, 9:335–372, 2002.
11. H. Riis Nielson, F. Nielson: Flow Logic: a multi-paradigmatic approach to static analysis. *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, pages 223–244, Springer Verlag, 2002.

# Using Constraints To Analyze And Generate Safe Capability Patterns

Fred Spiessens, Yves Jaradin, and Peter Van Roy

Université catholique de Louvain  
Louvain-la-Neuve, Belgium  
{fsp,yjaradin,pvr}@info.ucl.ac.be

**Abstract.** We present a dual purpose CCP application for capability based security. In a first setting, the application analyzes capability patterns of collaboration by calculating upper bounds on the propagation of (overt) causal influence. In this setting, all the relied upon restrictions in the behavior of the subjects in the pattern are input and transformed into constraint propagators.

In a second setting, the application calculates how the behavior of a (set of) trusted subject(s) in the pattern should be restricted, given the global safety properties that have to be respected.

From earlier theoretical results [SV05], we are confident that our approach is complete (all safety breaches are found and the proposed restrictions on behavior are sufficient). Because the tool is currently in a very early stage of its implementation, we only present a small set of preliminary quantitative results.

## 1 Introduction

In 1976 Harrison, Ruzzo, and Ullman [HRU76] showed that the calculation of safety properties in general is an intractable problem. As their modeling language was Turing-complete, this intractability was the inevitable price to pay for its expressive power. A few years later, Take-Grant systems [BS79] were proposed for the analysis of capability based security problems [DH65]. In this model and its extensions, the safety properties are tractable [LS77,FB96], but the formalism lacks the power to express *carefully restricted* collaborative behavior. The need for a more expressive model that takes such restrictions into account is explained in [MS03,SV05].

The formal models presented in [SV05] provide the necessary expressive power to precisely model restrictions in subject behavior relevant to the propagation of information and authority between (sets of) collaborating entities. For finite configurations the calculation of the safety properties is tractable. The propagation induced by newly created entities is safely approximated by accumulating their behavior into the creating entity (parent). This result allows us to safely model unknown (untrusted) entities without considering their possible offspring. When modeling an entity's behavior, only the behavior restrictions it shares with all its potential children will be modeled as actual restrictions.

In the tool we present here, subject creation is restricted to this *implicit* form. The development is currently in a prototype stage and has lots of other limitations that will gradually be removed as the tool matures. Only the propagation of subjects is currently supported, and data propagation can only be modeled by substituting the data with non-collaborative subjects.

Section 1.1 introduces the most important capability security concepts. Section 2 gives a birds-eye overview of the tool. Design and implementation details are discussed in Section 3. Section 4 shows an example of how the tool can be used. The most important future extensions are listed in Section 5.

## 1.1 Glossary

Before explaining the goal and the approach of the tool, let us clarify the most important terms that will be used in this paper:

**Entity :** A loaded instance of a programmed entity like a procedure, an object, a process, a component or an agent. An entity can only be accessed (used) via unforgeable references (*capabilities*) that combine the *designation* of the entity with the *authority* to use the entity.

**Subject :** The modeled representation of an entity (possibly representing also the set of entities created at runtime by the entity, as explained earlier). Subjects could for instance be modeled from static analysis. Alternatively, subjects can be *specifications* for entities yet to be programmed (e.g. model based programming).

**Subject behavior :** The *willingness* of a subject to collaborate with another subject in a certain way. A subject's behavior should be a safe (over-) approximation of the behavior of the entity it models. If only the slightest possibility of collaboration exists that can lead to the entity propagating information or authority, the corresponding subject should have this behavior. There is more on subject behavior in section 2.2.

**Potential Authority :** The possible effects on propagation of authority and information that *could* be exerted by an entity if the entity *would be programmed* to do so. A subject's potential authority is a safe (over-)approximation of the potential authority of the entity it models.

**Actual Authority :** The possible effects on propagation of authority and information that can be exerted by an entity, when we take into account what is known about *its actual behavior*. A subject's actual authority is a safe (over-)approximation of the actual authority of the entity it models.

**Capability rules of propagation:** In pure capability systems, propagation of authority and information is only possible via either:

1. **collaboration :** an entity (the invoker, indicated by the prefix *i*) can initiate collaboration with another entity it has access to (the responder, indicated by the prefix *r*). In such a collaboration, either of them (the emitter) can provide data or subjects it has access to, to the other (the collector) if the latter is willing to collaborate in that way. It is always the emitter who decides what data or authority will be propagated, and it is always the invoker who decides what entity to collaborate with.

2. **parenthood** : New entities can only be created by entities. An entity that creates a new entity thereby gets the sole access to it.
  3. **endowment** : The parent entity, upon creation of its child, endows the child with a subset of its own access.
- Capability systems and their rules for (overt) propagation of influence are described in [MS03,SV05]).
- Configuration** : An access graph of subjects. A configuration can evolve via collaboration between its subjects. Such collaboration is governed by the capability rules of propagation and by the behavior of the subjects.
- Capability Pattern** : A useful configuration together with its well understood and described safety properties (access that is prevented) and liveness properties (access that is *not* prevented).

## 2 Overview of the Tool

### 2.1 The Goals

Figure 1 depicts the Caretaker pattern that will be a running example throughout this paper. *Caretaker*, created and controlled by *Alice*, is a proxy for *Carol*, to be used by *Bob*. *Alice* can order *Caretaker* to stop collaborating, in an attempt to revoke the *authority-to-invoke-Carol* she has granted to *Bob*. The fact that *Bob* and *Dave* are undefined subjects (modeling unknown entities and their offspring) is indicated by a shadow.

For the pattern to really allow revocation, *Bob* should never get direct access to *Carol* (indicated with the dashed arrow ending in a cross). Because the pattern is to be useful in a non-trivial context, we want to also make sure that *Bob* will not be prevented from getting access to *Dave* (indicated by the dashed arrow from *Bob* tot *Dave*).

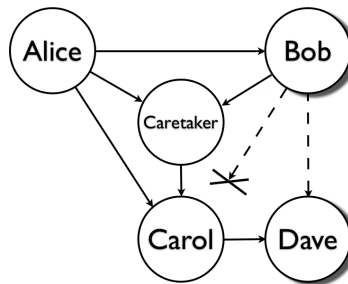


Fig. 1. The Caretaker Pattern for Revocation

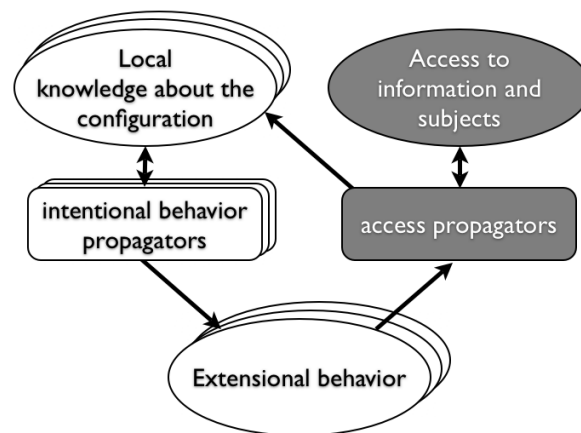
In this paper, we use the term *safety property* to mean access that is effectively prevented, while *liveness property* refers to access that is *not* prevented by restrictions in the behavior of the subjects. The tool can be used for two complementary goals:

1. **Check Requirements :** From the specifications for the behavior of the trusted collaborating entities, the maximal propagation of access is calculated. The result will show directly whether or not the required safety and liveness properties are satisfied. In the example, the behavior of *Alice*, *Caretaker* and *Carol* (the trusted subjects) will be decisive.
2. **Calculate Behavior Restrictions :** Given a set of required safety- and liveness properties, what minimal sets of behavior restrictions for a certain subject can ensure the global properties? The subject(s) of which the behavior restrictions are calculated will be called query-subject(s). For multiple query subjects the minimally restrictive combinations of necessary restrictions in the behavior of the query subjects will be calculated.

In the example, the minimal sets of behavior restrictions for *Carol* can be calculated if the behaviors of *Alice* and *Caretaker* are fixed. Alternatively one could for instance calculate all possible combinations of *Alice's* and *Carol's* restrictions, given the proxy-behavior of *Caretaker*. The current version of the tool is not ready to calculate combinations of three query subjects in a reasonable amount of time. It is possible to give a minimum behavior to a query subject though, which drastically speeds up the calculation of the remaining restrictions.

## 2.2 Monotonic and Confluent Approach

All access to subjects (and to data) that is present in an initial configuration will be stored as simple boolean constraints in a constraint store. Every subject's initial knowledge about the configuration (its relations towards subjects and data it has access to) will also be presented as boolean constraints.



**Fig. 2.** The propagation of access, knowledge and behavior

A set of subject-specific propagators corresponding to the subject's intentional behavior will test (*ask*) the subject's current knowledge and generate new behavior constraints in the constraint store (boolean constraints). We call these constraints the subject's extensional behavior. Another set of propagators will model how access can propagate via collaboration in the configuration – generating new access constraints – and will also make sure that the collaborating subjects are informed of the observable effects of the collaboration, generating extra knowledge constraints.

The working of these propagators, from the point of view of a single subject, are graphically depicted in Figure 2. The system-wide propagators and constraints are gray. The constraint store will eventually reach a fix point, representing the maximum possible propagation of access in the configuration, from which goal **1.** can be inferred. If goal **2.** is pursued, a distribute-and-search process will search for (one or all) satisfactory solution(s) to the query-subject's extensional behavior.

The access constraints have the form  $access(S_1, X)$  meaning that subject  $S_1$  has access to  $X$  ( $X$  being another subject or data). The extensional behavior constraints of a subject  $S_1$  have the following form ( $S_1$  is an *implicit* first argument that will be made *explicit* when used by the access propagators) :

predicate	meaning
$iEmit(S_2, X)$	Subject $S_1$ is willing to invoke subject $S_2$ , and pass $X$ as a parameter to it.
$iCollect(S_2)$	Subject $S_1$ is willing to invoke subject $S_2$ , and accept whatever $S_2$ provides as a return value
$rEmit(X)$	Subject $S_1$ is willing to return subject $X$ upon being invoked.
$rCollect()$	Subject $S_1$ is willing to accept whatever input argument $S_1$ is being invoked with.
$rExchange(X, Y)$	Subject $S_1$ is willing to accept whatever input argument $S_1$ is being invoked with, and if it is $X$ , then it is willing to return $Y$ .

Notice that  $rExchange()$  allows  $S_1$  to differentiate its behavior towards its invokers, based on a proof-of-access that these invokers can provide to  $S_1$ .

The knowledge constraints generated by the access propagators toward subject  $S_1$  have the following form ( $S_1$  is now an *explicit* first argument that will be made *implicit* when used by subject  $S_1$ 's behavior propagators) :

predicate	meaning
$iEmitted(S_1, S_2, X)$	Subject $S_1$ has successfully invoked $S_1$ , and passed $X$ as a parameter to it.
$iCollected(S_1, S_2, Y)$	Subject $S_1$ has successfully invoked subject $S_2$ , and accepted $Y$ as a return value from the invocation. This means $S_1$ has now got access to $Y$
$rEmitted(S_1, X)$	Subject $S_1$ has successfully returned subject $X$ upon being invoked.
$rCollected(S_1, Y)$	Subject $S_1$ has accepted input argument $Y$ upon being invoked. $S_1$ has now got access to $Y$ .
$rExchanged(S_1, X, Y)$	Subject $S_1$ has returned $Y$ on the basis of having received $X$ <i>in the same invocation</i> .
$access(S_1, X)$	Subject $S_1$ has access to $X$ , either acquired by collecting or from initial conditions.

The other knowledge constraints are subject-specific, and only the subject's intentional behavior propagators will be able to read/write to them. They will have the subject itself as an *implicit* first argument.

### 2.3 Input

The tool takes an initial configuration as input, consisting of an access graph of named subjects of which the intentional behavior is described. The intentional behavior of every subject is given as a set of logical implications (Horn clauses). The condition (body) of such an implication will contain knowledge-predicates, the conclusion (head) can contain subject-specific knowledge predicates and extensional behavior predicates.

For example, the proxy behavior that will characterize *Caretaker* in the Caretaker pattern could be specified using a subject-specific knowledge predicate  $isMyProxy()$  in the following rules:

$$\begin{aligned}
 iEmit(S, X) &:- isMyProxy(S) \wedge rCollected(X) \\
 iCollect(S) &:- isMyProxy(S) \\
 rEmit(X) &:- iCollected(S, X) \\
 rCollect() &
 \end{aligned}$$

Subjects are further initialized with a set of facts, that represent their initial partial knowledge (predicates) of the configuration. A subject's initial knowledge predicates will typically represent part of its relations towards the subjects (or data) it initially has access to. The access graph is also described as a set of (access) facts.

A set of safety properties and liveness properties are added to the configuration in the form of logical combinations of basic constraints (typically access constraints). The safety properties will be negated before being converted into a propagator that will cause failure upon possible violation of the property. Before a solution is validated, the liveness properties will also be verified.

If the goal is to calculate extensional behavior, the list of query subjects should be provided too.



## 2.4 Output

The tool calculates from the initial configuration, the maximal configuration containing all possible access. When a failure is detected, it is straightforward to construct witness traces (evidence) of how the safety properties can be violated, from the constraints that were added to the store. Upon success, the store shows the maximal extent to which data and capabilities (subjects) can be propagated. It is then simple to check the liveness requirements.

If the extensional behavior of a query subject is calculated (via search), the tool extracts for every solution, from the quiescent store corresponding to that solution, an overview of its extensional behavior predicates that are true (effectively leading to allowed collaboration), false (collaboration could lead to a violated safety property), or undefined (not relevant in the configuration).

## 3 CCP Based Implementation

In this section we describe how the constraint propagators are designed. Apart from the propagators for intentional subject behavior and access propagation, we present some additional propagators that will assist the calculation. We also describe our strategy for search and distribution.

### 3.1 Constraint Propagators

**Propagators for Access** These propagators are independent of the actual configuration and the specified behavior of the subjects. They are a direct representation of the way how, in capability systems, information and access are propagated via collaboration.

1. Granting: the invoker emits, the responder collects.

$$\frac{access(S_1, S_2) \wedge access(S_1, X) \wedge iEmit(S_1, S_2, X), \wedge rCollect(S_2)}{access(S_2, X) \wedge iEmitted(S_1, S_2, X) \wedge rCollected(S_2, X)} \quad (1)$$

2. Take rule: the invoker collects, the responder emits

$$\frac{access(S_1, S_2) \wedge access(S_2, X) \wedge iCollect(S_1, S_2), \wedge rEmit(S_2, X)}{access(S_1, X) \wedge iCollected(S_1, S_2, X) \wedge rEmitted(S_2, X)} \quad (2)$$

3. Exchange rule: both invoker and responder emit and collect. The responder bases his decision to emit on (obtainable knowledge about) what he collected during the invocation.

$$\frac{access(S_1, S_2) \wedge access(S_1, X) \wedge access(S_2, Y) \wedge iEmit(S_1, S_2, X) \wedge rCollect(S_2) \wedge iCollect(S_1, S_2) \wedge rExchange(S_2, X, Y)}{access(S_2, X) \wedge access(S_1, Y) \wedge iEmitted(S_1, S_2, X) \wedge rCollected(S_2, X) \wedge iCollected(S_1, S_2, Y) \wedge rExchanged(S_2, X, Y)} \quad (3)$$

Using the propagators (1) and (2) we can reduce (3) to:

$$\frac{iEmitted(S_1, S_2, X) \wedge access(S_2, Y) \wedge iCollect(S_1, S_2) \wedge rExchange(S_2, X, Y)}{access(S_1, Y) \wedge iCollected(S_1, S_2, Y) \wedge rExchanged(S_2, X, Y)} \quad (4)$$

**Behavior Propagators** These represent the subject-specific reaction to positive knowledge about access to subjects and data, and about the way this access was acquired. They can refine knowledge and use knowledge to generate behavior. They are restricted in the sense that they cannot generate new access (that would defy the capability rules) or knowledge of the kind that is produced by the access propagators. To reflect the fact that subjects can only refine their own knowledge, and generate their own behavior, these propagators will also be restricted in scope. The first argument of every predicate is *implicit* and designates the subject who's behavior is being described. It will become *explicit* only for the access-propagators and for the assisting propagators.

A Horn clause that partially describes subject  $S_1$ 's behavior like this:

$$behavior(B_1, \dots B_n) \leftarrow condition_1(C_{1,1} \dots C_{1,k}) \wedge \dots \wedge condition_j(C_{j,1}, \dots C_{j,m}) \quad (5)$$

... will be converted into a propagator like this:

$$\frac{condition_1(S_1, C_{1,1}, \dots C_{1,k}) \wedge \dots \wedge condition_j(S_1, C_{j,1}, \dots C_{j,i})}{behavior(S_1, B_1, \dots B_n)} \quad (6)$$

The behavior for an unknown (untrusted) subject  $S_1$  can be represented with a single propagator:

$$\frac{true}{iCollect(S_1, S_2) \wedge iEmit(S_1, S_2, X) \wedge rCollect(S_1) \wedge rEmit(S_1, X)} \quad (7)$$

**Assisting Propagators** As soon as one of two "unknown" (completely collaborative) subjects has direct access to the other one, they will inevitably end up sharing the same access. Therefore, the query subject should not even consider treating these subjects differently, as it will not have a different effect. For every pair of unknown subjects,  $(S_1, S_2)$ , and for every query subject  $S_q$ , we will add the following propagators:

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iEmit(S_q, S_1, X) = iEmit(S_q, S_2, X)} \quad (8)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iEmit(S_q, S, S_1) = iEmit(S_q, S, S_2)} \quad (9)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{iCollect(S_q, S_1) = iCollect(S_q, S_2)} \quad (10)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rEmit(S_q, S_1) = rEmit(S_q, S_2)} \quad (11)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rExchange(S_q, S_1, X) = rExchange(S_q, S_2, X)} \quad (12)$$

$$\frac{access(S_1, S_2) \vee access(S_2, S_1)}{rExchange(S_q, X, S_1) = rExchange(S_q, X, S_2)} \quad (13)$$

Without going into details about the implementation, it is easy to see how these propagators can be efficiently implemented: as far as the query subjects are concerned, their extensional behavior can consider both subjects as one aggregate subject. This principle can also be used in a weaker form for any two subjects, when it would be useless for the query subjects to differentiate (a particular part of) their behavior towards the one or the other. It is a form of symmetry braking in the constraint model.

Safety properties are mere boolean constraints that are set to *false*, to cause failure when they are unified with *true* by a propagator. We are experimenting with propagators for safety properties in the form (14) and (15), to promote early failure detection. Propagating *false* to a query subject's extensional behavior constraints will decrease the depth of the search tree.

$$\frac{\neg access(S_q, X)}{\neg(access(S_1, S_q) \wedge access(S_1, X) \wedge iEmit(S_1, S_2, X) \wedge rCollect(S_2))} \quad (14)$$

$$\frac{\neg access(S_q, X)}{\neg(access(S_q, S_1) \wedge access(S_1, X) \wedge rEmit(S_1, X) \wedge iCollect(S_1, S_2))} \quad (15)$$

### 3.2 Constraint Implementations

We implement the tool in the Mozart environment [Moz03] for the multi-paradigm language Oz [Smo95,VH04], which provides strong support for concurrent constraint programming [Sch02]. Because the implementation of the basic boolean constraints can have a big impact on the efficiency of the propagators mentioned above, we are currently experimenting with two approaches in parallel, one using finite domain integers and the other one using finite sets of integers. We give a short description of both.

**Finite Domain Integer Constraints** Every predicate is modeled as a finite domain integer variable in a domain ranging from 0 (*false*) to 1 (*true*). Logical connectives can now be implemented as a product (logical and) or as a sum (logical or, using the appropriate domain for the sum). Propagator (16) shows how we implement the safety property propagator (14) with finite domain integers and the *sum* and *<*: (strictly smaller) propagators.

$$\frac{\neg access(S_q, X)}{sum([access(S_1, S_q), access(S_1, X), iEmit(S_1, S_2, X), rCollect(S_2)]) <: 4} \quad (16)$$

To avoid a combinatorial explosion of the number of finite domain variables, we implement logical *and* with nested implications *impl* where appropriate. The implication propagator will wait for its condition to be true, before telling its conclusion. The conclusion can again be an implication. This is how we implement the access propagators in this approach. Propagator (17) gives an example of how the granting propagator (1) is implemented.

$$\begin{aligned} & \text{impl}(\text{access}(S_1, S_2), \\ & \quad \text{impl}(\text{access}(S_1, X), \\ & \quad \quad \text{impl}(\text{iEmit}(S_1, S_2, X), \\ & \quad \quad \quad \text{impl}(\text{rCollect}(S_2), \\ & \quad \quad \quad \quad (\text{access}(S_2, X) \wedge \text{iEmitted}(S_1, S_2, X) \\ & \quad \quad \quad \quad \quad \wedge \text{rCollected}(S_2, X) \quad )))) \end{aligned} \quad (17)$$

**Finite Sets Constraints** In this approach we present an  $n$ -ary predicate as the finite set of all  $n$ -tuples of subjects that satisfy the predicate. We assign a unique integer to each  $n$ -tuple of subjects, to represent that tuple in the set. Implications over predicates are translated into set inclusions over the corresponding finite sets of integers, disjunction is translated to union, and conjunction becomes intersection. Of course, these operations should only be performed on compatible predicates.

In (18) and (19) we consider two clauses in *Caretaker's* behavior to explain how the predicates are made compatible.

$$\text{iEmit}(S, X) :- \text{isMyProxy}(S) \wedge \text{rCollected}(X) \quad (18)$$

$$\text{rEmit}(X) :- \text{iCollected}(S, X) \quad (19)$$

In the body of clause (18) we cannot simply use set intersection, because  $\text{isMyProxy}(S)$  and  $\text{rCollected}(X)$  have a different variable. First we have to make the cartesian product in the following way:

$$\text{iEmit}(S, X) :- (\text{isMyProxy}(S) \times \text{isSubj}(X)) \wedge (\text{isSubj}(S) \times \text{rCollected}(X))$$

We use  $\text{isSubj}()$  as a unary predicate that is true for every subject. The clause now translates to the finite set propagator:

$$\text{iEmit} \subseteq (\text{isMyProxy} \times \text{isSubj}) \cap (\text{isSubj} \times \text{rCollected}).$$

The cartesian product of finite sets is implemented by recalculating the individual integers (tuples) of the result set.

The head of clause (19) has one less variable than its body. Therefore we use a projection operation  $P_{(\dots)}$  that extracts a sub-tuple from every tuple in the predicate, and we convert the clause to:  $\text{rEmit}(X) :- P_{(X)}(\text{iCollected}(S, X))$ . This translates to the finite set propagator:  $\text{rEmit} \subseteq P_{(2)}(\text{iCollected})$ .

All clauses can thus be translated to finite set propagators using the proper combination of cartesian product, projection, inclusion, union, and intersection. Because the cartesian product is the most costly operation, we try to minimize its use and the size of its argument sets. For instance, the actual implementation of the clause (19) will be simplified to:  $\text{iEmit} \subseteq \text{isMyProxy} \times \text{rCollected}$ .

**Preliminary Comparison** The current state of the tool does not yet allow us to make quantified comparisons or draw conclusions about which of the two approaches is best suited for what kind of problems. We provide for both approaches the preliminary benchmark results for the calculation of *Carol's* restriction in the caretaker pattern as described in Section 4. We currently believe that the optimal overall approach will be a merge of both.

The finite domain integers approach finds the 4 solutions in 5 seconds, using 318 search nodes (not failed neither succeeded nodes), with a search tree depth of 35.

The finite set approach finds the first three solutions after 1, 20, and 63 seconds (total time) respectively. The forth solution was not yet found after 30 minutes. The finite set approach finds the first tree solutions using 440, 6000, and >12400 search nodes with a maximal search tree depth of 37.

All calculations were done on a 1.25 GHz PowerPC G4 with 1 GB memory.

### 3.3 Search and Distribution Strategies

When testing suitable behavior for query subjects, we use a depth first search strategy. To detect failures as fast as possible, we order the extensional behavior constraints of the query subject(s) by the number of concurrent constraint propagators that are currently waiting for that basic constraint to become determined. This functionality is implemented in the `FD.reflect.nbSusps` built-in procedure. The distribution stops when no more behavior aspects have at least one propagator waiting for it, indicating that all feasible ways for propagating access have been exhausted.

To detect the maximal solutions first, we always try the 1-alternative (*true*) first (indicating willingness to collaborate).

Further symmetry breaking is done via a particular use of the branch-and-bound facilities provided by the environment (with our thanks to Raphaël Collet for pointing out this possibility). Whenever we find a solution, we add it to a list of currently found solutions, and then tell a constraint that the next solution should not be a sub-solution of any solution in the list. Sub-solutions are solutions with less-than-maximal relevant collaboration properties. The branch-and-bound constraint works like this:

```
proc{NoSubSolutions Recent Next}
  Recent.oldSols := (Recent.solution) | @(Recent.oldSols)
  {ForAll @(Recent.oldSols)
    proc {$ Traces#_}
      {FD.sum {Map {Filter Traces fun {$ Tr} Tr.value==0 end}
        fun {$ Tr} {GetPred Tr.subjId Tr.pred Next} end}
        ^>: ^ 0}
      end}
  end
```

The procedure `NoSubSolutions` adds a propagator that ensures that there will be at least one non-collaborative (= 0) relevant behavior predicate of every previous solution that will be collaborative (= 1) in the next solution. Together

with the choice strategy “try the 1-alternative first”, this ensures that no sub-solutions are found or searched for.

#### 4 Example

As an example we calculate *Carol's* necessary behavior restrictions in the “caretaker” pattern, introduced in Figure 1 of Section 2.

Since we don’t know the behavior of *Bob* and *Denis*, the only safe approximation is to consider them to be completely collaborative (unknown) subjects. The intentional behavior of *Alice* and *Caretaker* is listed in table 1, together with their initial access and knowledge.

**Table 1.** The behavior of *Alice* and *Caretaker*

<b>Alice</b>	
$iEmit(S, X) :- use(S) \wedge pass(X)$ $iEmit(S, X) :- isBob(S) \wedge isCaretaker(X)$ $iCollect(S) :- use(S) \wedge pass(S)$ $rEmit(X) :- pass(X)$ $rCollect() :- true$	knowledge $\rightarrow$ behavior
$pass(X) :- rCollected(X)$ $use(X) :- isCarol(X)$	knowledge $\rightarrow$ knowledge
$access(1) \wedge isSelf(1) \wedge use(1) \wedge pass(1)$ $access(2) \wedge isBob(2)$ $access(3) \wedge isCaretaker(3)$ $access(4) \wedge isCarol(4)$	initial knowledge
<b>Caretaker</b>	
$iEmit(S, X) :- isMyProxy(S) \wedge rCollected(X)$ $iCollect(S) :- isMyProxy(S)$ $rEmit(X) :- iCollected(S, X)$ $rCollect()$	knowledge $\rightarrow$ behavior
$access(3) \wedge isSelf(3)$ $access(4) \wedge isMyProxy(4)$	initial knowledge

Table 2 lists the solutions found for *Carol's* extensional behavior.

The first two solutions restrict *Carol's* behavior towards Alice, because *Alice* could inadvertently allow *Carol* to be collected from here by *Bob*. The precautions taken in *Alice's* behavior do not exclude this: the sixth clause in *Alice's* behavior shows that when she collects *Carol* upon being invoked, she will pass her on.

The last two solutions are not very interesting, since they don’t allow *Carol* to accept any capabilities upon being invoked. An inspection of the store showed us that in these two cases, *Alice* (rather than *Carol*) is responsible for providing *Bob* access to *Denis* (the liveness property).

**Table 2.** solutions

1	Carol should not <i>rEmit</i> herself. Carol should not <i>iEmit</i> herself to Alice, Bob, or Denis.
2	Carol should not <i>rEmit</i> herself or Alice Carol should not <i>iEmit</i> herself to Bob or Denis, Carol should not <i>iEmit</i> Alice to Bob or Denis.
3	Carol should not <i>rEmit</i> herself Carol should not <i>iEmit</i> herself to Denis, Carol should not <i>iCollect</i> from Denis. Carol should not <i>rCollect</i> .
4	Carol should not <i>rEmit</i> herself Carol should not <i>iEmit</i> herself to Bob or Denis Carol should not <i>rCollect</i> .

The *exchange()* predicate was not used in the example, because we did not yet have a stable and reliable implementation for it.

## 5 Future Extensions

### Expressive Power

**Adding data :** We will soon add support for data. The current solution uses non-cooperative subjects as data and does not allow to reason about the more unexpected and indirect ways in which information can flow. For instance, when a client can influence the behavior of a server, and that behavior is visible to another client, information can flow from one client to the other. Our theoretical model allows to reason about this kind of data flow, and so should our tool.

**Exchange :** *rExchange()* is a recent addition of which we have to explore the possibilities and limitations. We would like to use it for enabling the *Caretaker* proxy to decide its collaboration per invocation, but invocation-based granularity can easily lead to a combinatorial explosion.

**Derived safety properties :** It is better to reason about the *flow* of authority than only about the *effects* of authority propagation. This can be done in flow-graphs (with arcs representing the direction of the flow) that are derived from the configuration. We will use reachability constraints [QVD05] in derived flow graphs to express more elaborated safety properties. We will then be able to express the more precise safety property for the caretaker pattern: "Carol's authority should be *reachable* for Bob only via the caretaker".

### Functionality

**Calculate intentional behavior :** To provide a real specification for the query subject's intentional behavior we have to derive such specifications from the extensional behavior of the query subjects.

**Real pattern generation :** We want to experiment with adding trusted subjects to a configuration in critical places, when no safe solutions can be found in a pattern. This would allow us to generate patterns from high-level specifications.

### Performance and Scalability

**Add pruning :** The propagators for the safety-properties should help pruning the search space more than they do now.

**Merging the two approaches :** The approach based on finite sets has not yet been optimized to the level of the finite integer domain approach. We need to take care of that, and then measure the performance for different kinds of problems to find out which of the two approaches in Section 3.2 is the most performing and scalable, and how we can merge them to get the best of both worlds.

### User Interface

**Write a parser :** Currently we input the problems directly in parsed form.

**Web interface :** The tool would be useful to the community of capability developers. Therefore we want to wrap it into a web application.

**Integrating GraphViz :** We have an ad-hoc connection to the GraphViz tool [GN00,KN93] for visualizing the graphs generated by the solutions. We will properly integrate this visualization tool.

## 6 Related Work

Whereas actual applications of CCP to security are not widespread yet, we see a few interesting opportunities that are related to model checking and pattern generation as we describe it in our paper.

The work of Joshua Guttman et al. [Jos05] uses a datalog-like language for secure protocol design . We believe that that by using constraints (and search) the way we do, that approach could be enriched to also support the “generation” of such protocols, from general descriptive rules.

Jan Jürjen’s work on security specifications in UML [Jö5] – again a model-based approach – could probably also benefit from extensions with constraint-based model checking.

## 7 Acknowledgments

This work was partially funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935, and partly by the MILOS project of the Walloon Region of Belgium under convention 114856. We thank Raphaël Collet for discussing the formal aspects of the model. We thank Mark Miller for his advice about capability-based security. We thank the reviewers for their useful comments and suggestions.



## References

- [BS79] Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 45–54. ACM Press, 1979.
- [DH65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.
- [FB96] Jeremy Frank and Matt Bishop. Extending the take-grant protection system, December 1996. Available at:  
<http://citeseer.ist.psu.edu/frank96extending.html>.
- [GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
- [HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.
- [J05] Jan Jürjens. *Secure Systems Development with UML*. Springer, Berlin, June 2005.
- [JM04] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Mathematics and Visualization. Springer, Dec 2004.
- [Jos05] Joshua D. Guttman and Jonathan C. Herzog and John D. Ramsdell and Brian T. Sniffen. Programming cryptographic protocols. Technical report, The MITRE Corporation, 2005. Available at  
<http://www.ccs.neu.edu/home/guttman/>.
- [KN93] Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot*. Murray Hill, NJ, 1993.
- [LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.
- [Moz03] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2003. Available at <http://www.mozart-oz.org/>.
- [MS03] Mark S. Miller and Jonathan Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.
- [QVD05] Luis Quesada, Peter Van Roy, and Yves Deville. The reachability propagator. Research Report INFO-2005-07, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.
- [Sch02] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.
- [Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
- [SV05] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in Capability-Based systems, 2005. To be published in *Lecture Notes in Computer Science* (Springer-Verlag). Available at  
<http://www.info.ucl.ac.be/people/fsp/tgc/tgc05fs.pdf>. Presentation at  
<http://www.info.ucl.ac.be/people/fsp/auredsysfinal.mov>.
- [VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.