Proceedings

Foundations of Computer Security

Affiliated with LICS'02


FLoC'02

Copenhagen, Denmark
July 25–26, 2002

Edited by
Iliano Cervesato

With support from

Office of Naval Research
International Field Office

Table of Contents

Verification of Security Protocols

Programming Language Security

Panel

Preface

Computer security is an established field of Computer Science of both theoretical and practical significance. In recent years, there has been increasing interest in logic-based foundations for various methods in computer security, including the formal specification, analysis and design of cryptographic protocols and their applications, the formal definition of various aspects of security such as access control mechanisms, mobile code security and denial-of-service attacks, and the modeling of information flow and its application to confidentiality policies, system composition, and covert channel analysis.

This workshop continues a tradition, initiated with the Workshops on Formal Methods and Security Protocols — FMSP — in 1998 and 1999 and then the Workshop on Formal Methods and Computer Security — FMCS — in 2000, of bringing together formal methods and the security community. The aim of this particular workshop is to provide a forum for continued activity in this area, to bring computer security researchers in contact with the FLoC community, and to give FLoC attendees an opportunity to talk to experts in computer security.

Given the affinity of themes, FCS was synchronized with the FLoC'02 Verification Workshop (VERIFY). Sessions with a likely overlap in audience were held jointly. Moreover, authors who thought their paper to be of interest for both FCS and VERIFY could indicate that it be considered a joint submission, and it was reviewed by members of both program committees.

FCS received 22 submissions, 10 of which were joint with VERIFY. The review phase selected 11 of them for presentation; 5 of these were joint with VERIFY. This unexpected number of papers lead to extending FCS by one day.

Many people have been involved in the organization of the workshop. John Mitchell, assisted by the Organizing Committee, is to be thanked for bringing FCS into existence as part of FLoC. The Program Committee did an outstanding job selecting the papers to be presented, in particular given the short review time. We are very grateful to the VERIFY chairs, Heiko Mantel and Serge Autexier, for sharing the organizational load and for the numerous discussions. Sebastian Skalberg, Henning Makholm and Klaus Ebbe Grue, our interface to FLoC, turned a potential bureaucratic nightmare into a smooth ride. Finally we are grateful to the authors, the panelists and the attendees who make this workshop an enjoyable and fruitful event.

Iliano Cervesato

FCS'02 Program Chair

Workshop Committees

Program Committee

Iliano Cervesato (chair), ITT Industries, USA
Véronique Cortier, ENS Cachan, France
Grit Denker, SRI International, USA
Carl Gunter, University of Pennsylvania, USA
Alan Jeffrey, DePaul University, USA
Somesh Jha, University of Wisconsin — Madison, USA
Trevor Jim, AT&T Labs, USA
Heiko Mantel, DFKI Saarbrücken, Germany
Catherine Meadows, Naval Research Laboratory, USA
Flemming Nielson, Technical University of Denmark
Birgit Pfitzmann, IBM Zürich, Switzerland
David Sands, Chalmers University of Technology, Sweden
Stephen Weeks, InterTrust, USA

Organizing Committee

Martín Abadi, University of California — Santa Cruz, USA
Hubert Comon, ENS Cachan, France
Joseph Halpern, Cornell University, USA
Gavin Lowe, Oxford University, UK
Jonathan K. Millen, SRI International, USA
Michael Mislove, Tulane University, USA
John Mitchell (chair), Stanford University, USA
Bill Roscoe, Oxford University, UK
Peter Ryan, University of Newcastle upon Tyne, UK
Steve Schneider, Royal Holloway University of London, UK
Vitaly Shmatikov, SRI International, USA
Paul Syverson, Naval Research Laboratory, USA
Michael Waidner, IBM Zürich, Switzerland
Rebecca Wright, AT&T Labs, USA

Session I

Foundations of Security

On the Decidability of Cryptographic Protocols with Open-ended Data Structures

Ralf Küsters

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel, Germany
kuesters@ti.informatik.uni-kiel.de

Abstract

Formal analysis of cryptographic protocols has mainly concentrated on protocols with closed-ended data structures, where closed-ended data structure means that the messages exchanged between principals have fixed and finite format. However, in many protocols the data structures used are open-ended, i.e., messages have an unbounded number of data fields. Formal analysis of protocols with open-ended data structures is one of the challenges pointed out by Meadows. This work studies decidability issues for such protocols. We propose a protocol model in which principals are described by transducers, i.e., finite automata with output, and show that in this model security is decidable and PSPACE-hard in presence of the standard Dolev-Yao intruder.

1 Introduction

Formal methods are very successful in analyzing the security of cryptographic protocols. Using these methods, many flaws have been found in published protocols. By now, a large variety of different methods and tools for cryptographic protocol analysis is available (see [17] for an overview). In particular, for different interesting classes of protocols and intruders, security has been shown to be decidable, usually based on the Dolev-Yao model [7] (see the paragraph on related work).

Previous work has mostly concentrated on protocols with *closed-ended data structures*, where messages exchanged between principals have fixed and finite format. In what follows, we will refer to these protocols as *closed-ended protocols*. In many protocols, however, the data structures are *open-ended*: the exchanged messages may have an unbounded number of data fields that must be processed by a principal in one receive-send action, where receive-send action means that a principal receives a message and reacts, after some internal computation, by sending a message. One can, for example, think of a message that consists of an a priori unbounded sequence of requests and a server who needs to process such a message in one receive-send action; see Section 2 for concrete examples.

This paper addresses open-ended protocols, and thus, deals with one of the challenges pointed out by Meadows [17]. The goal is to devise a protocol model rich enough to capture a large class of open-ended protocols such that security is decidable; the long-term goal is to develop tools for automatic verification of open-ended protocols.

Open-ended protocols make it necessary to model principals who can perform in one receive-send action an unbounded number of *internal actions*; only then can they handle open-ended data structures. Therefore, the first problem is to find a good computational model for receive-send actions. It turns out that one cannot simply extend the existing models. More specifically, Rusinowitch and Turuani [21] describe receive-send actions by single rewrite rules and show security to be NP-complete. In this model, principals have unbounded memory. Furthermore, the terms in the rewrite rules may be non-linear, i.e., multiple occurrence of one variable is allowed, and thus, a principal can compare messages of arbitrary size for equality. To handle open-ended protocols, we generalize the model by Rusinowitch and Turuani in a canonical way and show that if receive-send actions are described by *sets* of rewrite rules, security is undecidable, even with i) finite memory and non-linear terms, or ii) unbounded memory and linear terms. Consequently, we need a computational model in which principals have finite memory and cannot compare messages of arbitrary size for equality.

For this reason, we propose to use transducers, i.e., finite automata with output, as the computational model for receive-send actions, since transducers satisfy the above restrictions — they have finite memory and cannot compare messages of arbitrary size for equality —, and still can deal with open-ended data structures. In Section 5.1 our so-called transducer-based model is discussed in detail. The main technical result of this paper is that in the transducer-based model, security is decidable and PSPACE-hard under the following assumptions: the number of sessions is bounded, i.e., a protocol is analyzed assuming a fixed number of interleaved protocol runs; nonces and complex keys are not allowed. We, however,

put no restrictions on the Dolev-Yao intruder; in particular, the message size is unbounded. These are standard assumptions also made in most decidable models for closed-ended protocols [21, 2, 13].[1] Just as in these works, the security property we study in the present paper is secrecy.

The results indicate that from a computational point of view, the analysis of open-ended protocols is harder than for closed-ended protocols, for which security is "only" NP-complete [21]. The additional complexity comes from the fact that now we have, beside the Dolev-Yao intruder, another source of infinite behavior: the unbounded number of internal actions (i.e., paths in the transducers of unbounded length). This makes it necessary to devise new proof techniques to show decidability. Roughly speaking, using that transducers only have finite memory we will use a pumping argument showing that the length of paths in the transducers can be bounded in the size of the problem instance.

Related work. All decidability and undecidability results obtained so far only apply to closed-ended protocols. Decidability depends on the following parameters: bounded or unbounded number of sessions, bounded or unbounded message size, absence or presence of pairing, nonces, and/or complex keys.

Usually, if one allows an unbounded number of sessions, security is undecidable [1, 8, 2, 9]. There are only a few exceptions: For instance, if the message size is bounded and nonces are disallowed, security is EXPTIME-complete [8]; if pairing is disallowed, security is in P [6, 2]. The situation is much better if one puts a bound on the number of sessions; from results shown by Lowe [15] and Stoller [23] it follows that, under certain conditions, one can assume such bounds without loss of generality. With a bounded number of sessions and without nonces, security is decidable even if pairing is allowed and the message size is unbounded [21, 2, 13]. In fact, in this setting, security is NP-complete, with [21] or without [2] complex keys. We make exactly the same assumptions in our models, where we use atomic keys.

To the best of our knowledge, the only contributions on formal analysis of open-ended protocols are the following: The recursive authentication protocol [5] has been analyzed by Paulson [19], using the Isabelle theorem prover, as well as by Bryans and Schneider [4], using the PVS theorem prover; the A-GDH.2 protocol [3] has been analyzed by Meadows [16] with the NRL Analyzer, and manually by Pereira and Quisquater [20], based on a model similar to the strand spaces model. As mentioned, decidability issues have not been studied so far.

Structure of the paper. In Section 2, we give examples of open-ended protocols. We then define a generic model for describing open-ended protocols (Section 3). In this model, receive-send actions can be arbitrary computations. In Section 4, we consider the instances of the generic model in which receive-send actions are specified by sets of rewrite rules, and show the mentioned undecidability result. The transducer-based model, the instance of the generic protocol model in which receive-send actions are given by transducers, is introduced in Section 5. This section also contains the mentioned discussion. In Section 6 the actual decidability and complexity results are stated. Finally, we conclude in Section 7.

Due to space limitations, in this paper we have largely omitted technical details and rather focused on the introduction and the discussion of our models. We only provide the proof ideas of our results. The full proofs can be found in the technical report [14]. It also contains a description of the recursive authentication protocol (see Section 2) in our transducer-based model.

2 Examples of Open-ended Protocols

An example of an open-ended protocol is the IKE Protocol [12], in which a principal needs to pick a *security association (SA)*, the collection of algorithms and other informations used for encryption and authentication, among an a priori unbounded list of SAs. Such a list is an open-ended data structure, since it has an unbounded number of data fields to be examined by a principal. An attack on IKE, found by Zhou [24] and independently Ferguson and Schneier [10], shows that when modeling open-ended protocols, the open-ended data structures must be taken into account, since otherwise some attacks might not be found. In other words, as also pointed out by Meadows [17], open-endedness is security relevant.

Other typical open-ended protocols are group protocols, for example, the recursive authentication protocol (RA protocol) [5] and the A-GDH.2 protocol [3], which is part of the CLIQUES project [22]. In the RA protocol, a key distribution server receives an a priori unbounded sequence of request messages (containing pairs of principals who want to share session keys) and must generate a corresponding sequence of certificates (containing the session keys). These sequences are open-ended data structures: In one receive-send action the server needs to process an unbounded number of data fields, namely the sequence of pairs of principals. Group protocols often allow an unbounded number of receive-send actions in one protocol run. In our models, we will, however, always assume a fixed bound on the number of receive-send actions, since otherwise, just as in the case of an unbounded number of sessions, security, in general, leads to undecidability. Nevertheless, even with such a fixed bound it is still necessary

[1] In [21, 13], however, complex keys are allowed.

to model open-ended data structures. In the RA protocol, a bound on the number of receive-send actions would imply that the sequence of requests generated by the principals is bounded. Nevertheless, the intruder can generate arbitrarily long request messages. Thus, the data structures are still open-ended, and the server should be modeled in such a way that, as in the actual protocol, he can process open-ended data structures.

In [14], we provide a formal description of the RA protocol in our transducer-based model.

3 A Generic Protocol Model

Our generic protocol model and the underlying assumptions basically coincide with the ones proposed by Rusinowitch et al. [21] and Amadio et al. [2] for closed-ended protocols. However, the important difference is that in the generic model, receive-send actions are, roughly speaking, binary relations over the message space, and thus can be interpreted as arbitrary computations. In the models of Rusinowitch et al. and Amadio et al. , receive-send actions are described by single rewrite rules or processes without loops, respectively.

Thus, the generic protocol model is a very general framework for open-ended protocols. In fact, it is much too general to study decidability issues. Therefore, in subsequent sections we will consider different instances of this model.

The main features of the generic protocol model can be summarizes as follows:

- a generic protocol is described by a finite set of principals;
- the internal state space of a principal may be infinite (which, for example, enables a principal to store arbitrarily long messages);
- every principal is described by a finite sequence of receive-send actions;
- receive-send actions are arbitrary computations.

We make the following assumptions:

- the intruder is the standard Dolev-Yao intruder; in particular, we do not put restrictions on the size of messages;
- principals and the intruder cannot generate new nonces, i.e., the nonces used in the analysis are only those already contained in the protocol description;
- keys are atomic;
- the number of sessions is bounded. More precisely, the sessions considered in the analysis are only those encoded in the protocol description itself.

These are standard assumptions also made in decidable models for closed-ended protocols. They coincide with the ones in [2], and except for complex keys, with those in [21, 13].

Let us now give a formal definition of the generic protocol model.

3.1 Messages

The definition of messages is rather standard. Let $\mathcal{N}$ denote a finite set of *atomic messages*, containing keys, names of principals, etc. as well as the special atomic message secret. The *set of messages* (over $\mathcal{N}$) is the least set $\mathcal{M}$ that satisfies the following properties:

- $\mathcal{N} \subseteq \mathcal{M}$;
- if $m, m' \in \mathcal{M}$, then $mm' \in \mathcal{M}$;
- if $m \in \mathcal{M}$ and $a \in \mathcal{N}$, then $\mathsf{enc}_a(m) \in \mathcal{M}$;
- if $m \in \mathcal{M}$, then $\mathsf{hash}(m) \in \mathcal{M}$.

As usual, concatenation is an associative operation, i.e., $(mm')m'' = m(m'm'')$. Note that we only allow for atomic keys, i.e., in a message $\mathsf{enc}_a(\cdot)$, a is always an atomic message.

Let ε denote the *empty message* and $\mathcal{M}_\varepsilon := \mathcal{M} \cup \{\varepsilon\}$ the set of messages containing ε. Note that ε is not allowed inside encryptions or hashes, that is, $\mathsf{enc}_a() \notin \mathcal{M}_\varepsilon$ and $\mathsf{hash}() \notin \mathcal{M}_\varepsilon$.

Later, we will consider terms, i.e., messages with variables. Let $V := \{v_0, \ldots, v_{n-1}\}$ be a set of variables. Then a *term* t (over V) is a message over the atomic messages $\mathcal{N} \cup V$, where variables are not allowed as keys, i.e., terms of the form $\mathsf{enc}_v(\cdot)$ for some variable v are forbidden. A *substitution* σ is a mapping from V into $\mathcal{M}_\varepsilon$. If t is a term, then $\sigma(t)$ denotes the message obtained from t by replacing every variable v in t by $\sigma(v)$.

The *depth* $\mathsf{depth}(t)$ *of a term* t is the maximum number of nested encryptions and hashes in t, i.e.,

- $\mathsf{depth}(\varepsilon) := 0$, $\mathsf{depth}(a) := 0$ for every $a \in \mathcal{N} \cup V$,
- $\mathsf{depth}(tt') := \mathsf{max}\{\mathsf{depth}(t), \mathsf{depth}(t')\}$,
- $\mathsf{depth}(\mathsf{enc}_a(t)) := \mathsf{depth}(t) + 1$,
- $\mathsf{depth}(\mathsf{hash}(t)) := \mathsf{depth}(t) + 1$.

3.2 The Intruder Model

We use the standard Dolev-Yao intruder model [7]. That is, an intruder has complete control over the network and can derive new messages from his current knowledge by composing, decomposing, encrypting, decrypting, and hashing messages. As usual in the Dolev-Yao model, we make the perfect cryptography assumption. We do not impose any restrictions on the size of messages.

The (possibly infinite) set of messages $\mathsf{d}(\mathcal{K})$ the intruder can derive from $\mathcal{K} \subseteq \mathcal{M}_\varepsilon$ is the smallest set satisfying the following conditions:

- $\mathcal{K} \subseteq \mathsf{d}(\mathcal{K})$;
- if $mm' \in \mathsf{d}(\mathcal{K})$, then $m \in \mathsf{d}(\mathcal{K})$ and $m' \in \mathsf{d}(\mathcal{K})$ (decomposition);
- if $\mathsf{enc}_a(m) \in \mathsf{d}(\mathcal{K})$ and $a \in \mathsf{d}(\mathcal{K})$, then $m \in \mathsf{d}(\mathcal{K})$ (decryption);
- if $m \in \mathsf{d}(\mathcal{K})$ and $m' \in \mathsf{d}(\mathcal{K})$, then $mm' \in \mathsf{d}(\mathcal{K})$ (composition);
- if $m \in \mathsf{d}(\mathcal{K})$, $m \neq \varepsilon$, and $a \in \mathcal{N} \cap \mathsf{d}(\mathcal{K})$, then $\mathsf{enc}_a(m) \in \mathsf{d}(\mathcal{K})$ (encryption);
- if $m \in \mathsf{d}(\mathcal{K})$ and $m \neq \varepsilon$, then $\mathsf{hash}(m) \in \mathsf{d}(\mathcal{K})$ (hashing).

3.3 Protocols

Protocols are described by sets of principals and every principal is defined by a sequence of receive-send actions, which, in a protocol run, are performed one after the other. Since we are interested in attacks, the definition of a protocol also contains the initial intruder knowledge. Formally, principals and protocols are defined as follows.

Definition 1 *A* generic principal Π *is a tuple* (Q, I, l, α) *where*

- Q *is the (possibly infinite) set of states of* Π*;*
- I *is the set of initial states of* Π*;*
- l *is the number of receive-send actions to be performed by* Π*;*
- α *is a mapping assigning to every* $j \in \{0, \ldots, l-1\}$ *a* receive-send action $\alpha(j) \subseteq Q \times \mathcal{M}_\varepsilon \times \mathcal{M}_\varepsilon \times Q$.

A generic protocol P *is a tuple* $(n, \{\Pi_i\}_{i<n}, \mathcal{K})$ *where*

- n *is the number of principals;*
- $\{\Pi_i\}_{i<n}$ *is a family of* n *generic principals, and*
- $\mathcal{K} \subseteq \mathcal{M}_\varepsilon$ *is the initial intruder knowledge.*

Note that receive-send actions are arbitrary relations. Intuitively, they take an input message (2. component) and nondeterministically, depending on the current state (1. component), return an output message (3. component) plus a new state (4. component). Later, when we consider instances of the generic protocol model, one receive-send action of a principal will consist of an unbounded number of internal actions. By allowing receive-send actions to be nondeterministic and principals to have a set of initial states, instead of a single initial state, one can model more flexible principals: for instance, those that nondeterministically choose one principal, who they want to talk to, or one SA from the list of SAs in the IKE Protocol.

We also remark that a protocol P is *not* parametrized by n. In particular, when we say that P is secure, we mean that P is secure given the n principals as defined in the protocol. We do not mean that P is secure for every number n of principals.

3.4 Attacks on Protocols

In an attack on a protocol, the receive-send actions of the principals are interleaved in some way and the intruder, who has complete control over the communication, tries to produce inputs for the principals such that from the corresponding outputs and his initial knowledge he can derive the secret message secret. Formally, an attack is defined as follows.

Definition 2 *Let* $P = (n, \{\Pi_i\}_{i<n}, \mathcal{K})$ *be a generic protocol with* $\Pi_i = (Q_i, I_i, l_i, \alpha_i)$*, for* $i < n$*. An* attack *on* P *is a tuple consisting of the following components:*

- *a total ordering* $<$ *on the set* $\{(i,j) \mid i < n, j < l_i\}$ *such that* $(i,j) < (i,j')$ *implies* $j < j'$ *(the execution order of the receive-send actions);*[2]
- *a mapping* ψ *assigning to every* (i,j), $i < n$, $j < l_i$, *a tuple*

$$\psi(i,j) = (q_i^j, m_i^j, {m'}_i^j, q_i^{j+1})$$

with

  - $q_i^j, q_i^{j+1} \in Q_i$ *(the state of* Π_i *before/after performing* $\alpha_i(j)$*); and*
  - $m_i^j, {m'}_i^j \in \mathcal{M}_\varepsilon$ *(the input message received and output message sent by* $\alpha_i(j)$*);*

such that

- $q_i^0 \in I_i$ *for every* $i < n$*;*
- $m_i^j \in \mathsf{d}(\mathcal{K} \cup \{{m'}_{i'}^{j'} \mid (i',j') < (i,j)\})$ *for every* $i < n, j < l_i$*;*
- $(q_i^j, m_i^j, {m'}_i^j, q_i^{j+1}) \in \alpha_i(j)$ *for every* $i < n, j < l_i$.

An attack is called successful *if* $\mathsf{secret} \in \mathsf{d}(\mathcal{K} \cup \{{m'}_i^j \mid i < n, j < l_i\})$.

The decision problem we are interested in is the following:

ATTACK: Given a protocol P, decide whether there exists a successful attack on P.

[2] Although, we assume a linear ordering on the receive-send actions performed by a principal, we could as well allow partial orderings (as in [21]) without any impact on the decidability and complexity results.

A protocol guarantees *secrecy* if there does not exist a successful attack. In this case, we say that the protocol is *secure*.

Whether ATTACK is decidable or not heavily depends on what kinds of receive-send actions a principal is allowed to perform. In the subsequent sections, we look at different instances of generic protocols, i.e., different computational models for receive-send actions, and study the problem ATTACK for the classes of protocols thus obtained.

4 Undecidability Results

We extend the model proposed by Rusinowitch and Turuani [21] in a straightforward way such that open-ended protocols can be handled, and show that this extension leads to undecidability of security.

The model by Rusinowitch and Turuani can be considered as the instance of the generic protocol model in which receive-send actions are described by single rewrite rules of the form $t \to t'$, where t and t' are terms.[3] The internal state of a principal is given implicitly by the values assigned to the variables occurring in the rewrite rules – different rules may share variables. In particular, a principal has unbounded memory to store information for use in subsequent receive-send actions. Roughly speaking, a message m is transformed by a receive-send action of the form $t \to t'$ into the message $\sigma(t')$, where σ is a substitution satisfying $m = \sigma(t)$. In [21], it is shown that in this setting, ATTACK is NP-complete.

Of course, in this model open-ended data structures cannot be handled since the left hand-side t of a rewrite rule has a fixed and finite format, and thus, one can only process messages with a fixed number of data fields.

A natural extension of this model, which allows to deal with open-ended data structures, is to describe receive-send actions by sets of rewrite rules, which can nondeterministically be applied to the input message, where, as in the model of Rusinowitch and Turuani, rewriting means top-level rewriting: If the rule $t \to t'$ is applied to the input message m yielding $\sigma(t')$ as output, another rule (nondeterministically chosen from the set of rules) may be applied to $\sigma(t')$. To the resulting output yet another rule may be applied and so on, until no rule is or can be applied anymore. The applications of the rules are the internal actions of principals. The instance of the generic protocol model in which receive-send actions are described by sets of rewrite rules as described above is called *rule-based protocol model*. In [14], we give a formal definition of this model. In this model, we distinguish between input, output, and process rules, and also put further restrictions on the rewrite rules such that they can be applied only a finite (but a priori unbounded) number of times.

[3] Since Rusinowitch and Turuani allow complex keys, the terms are more general than the ones we use here. However, we will only consider terms as defined in Section 3.1.

Theorem 3 *For rule-based protocols,* ATTACK *is undecidable.*

By reduction from Post's Correspondence Problem (PCP), this theorem is easy to show. It holds true, even for protocols consisting of only one principal, which may only perform one receive-send action. In other words, the undecidability comes from the internal actions alone.

However, the reduction does not work if only linear terms are allowed in rewrite rules. In linear terms, every variable occurs at most once, and therefore, one cannot compare submessages of arbitrary size for equality. Nevertheless, if principals can store one message and compare it with a submessage of the message being processed, we still have undecidability. Such protocols are called *linear-term one-memory protocols*; see [14] for the formal definition and the proof of undecidability, which is again by a rather straightforward reduction from PCP.

Theorem 4 *For linear-term one-memory protocols,* ATTACK *is undecidable.*

5 The Transducer-based Protocol Model

The previous section indicates that, informally speaking, when principals can process open-ended data structures and, in addition, can

1. compare submessages of arbitrary size (which is possible if terms are not linear), or
2. store one message and compare it with a submessage of the message being processed,

then security is undecidable. To obtain decidability, we need a device with only finite memory, and which does not allow to compare messages of arbitrary size. This motivates to use transducers to describe receive-send actions. In what follows, we define the corresponding instance of the generic protocol model. In Section 5.1, we will discuss capabilities and restrictions of our transducer-based model.

If Σ is a finite alphabet, Σ^* will denote the set of finite words over Σ, including the empty word ε.

Definition 5 *A* transducer $\mathcal{A}$ *is a tuple* $(Q, \Sigma, \Omega, I, \Delta, F)$ *where*

- Q *is the finite set of states of* $\mathcal{A}$*;*
- Σ *is the finite input alphabet;*
- Ω *is the finite output alphabet;*
- $I \subseteq Q$ *is the set of initial states of* $\mathcal{A}$*;*
- $\Delta \subseteq Q \times \Sigma^* \times \Omega^* \times Q$ *is the finite set of transitions of* $\mathcal{A}$*; and*

- *$F \subseteq Q$ is the set of final states of $\mathcal{A}$.*

A *path* π (of length n) in $\mathcal{A}$ from p to q is of the form $q_0(v_0, w_0)q_1(v_1, w_1)q_2 \dots (v_{n-1}, w_{n-1})q_n$ with $q_0 = p$, $q_n = q$, and $(q_i, v_i, w_i, q_{i+1}) \in \Delta$ for every $i < n$; π is called *strict* if $n > 0$, and v_0 and v_{n-1} are non-empty words. The word $v_0 \dots v_{n-1}$ is the *input label* and $w_0 \dots w_{n-1}$ is the *output label* of π. A path of length 0 has input and output label ε. We write $p(v, w)q \in \mathcal{A}$ ($p(v, w)q \in_s \mathcal{A}$) if there exists a (strict) path from p to q in $\mathcal{A}$ with input label v and output label w.

If $S, T \subseteq Q$, then $\mathcal{A}(S, T) := \{(p, v, w, q) \mid p \in S, q \in T, p(v, w)q \in \mathcal{A}\} \subseteq Q \times \Sigma^* \times \Omega^* \times Q$. The *output of $\mathcal{A}$ on input $v \in \Sigma^*$* is defined by $\mathcal{A}(v) := \{w \mid$ there exists $p \in I$ and $q \in F$ with $(p, v, w, q) \in \mathcal{A}(I, F)\}$.

If $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Omega \cup \{\varepsilon\}) \times Q$, $\mathcal{A}$ is called *transducer with letter transitions* in contrast to transducers with word transitions. The following remark shows that it suffices to consider transducers with letter transitions.

Remark 6 *Let $\mathcal{A} = (Q, \Sigma, \Omega, I, \Delta, F)$ be a transducer. Then there exists a transducer $\mathcal{A}' = (Q', \Sigma, \Omega, I, \Delta', F)$ with letter transitions such that $Q \subseteq Q'$, and $\mathcal{A}'(S, T) = \mathcal{A}(S, T)$ for every $S, T \subseteq Q$.*

In order to specify the receive-send actions of a principal, we consider special transducers, so-called message transducers, which satisfy certain properties. Message transducers interpret messages as words over the finite alphabet $\Sigma_{\mathcal{N}}$, consisting of the atomic messages as well as the letters "$\mathsf{enc}_a($", "$\mathsf{hash}($", and "$)$", that is,

$$\Sigma_{\mathcal{N}} := \mathcal{N} \cup \{\mathsf{enc}_a(\mid a \in \mathcal{N}\} \cup \{\mathsf{hash}(,\ )\}.$$

Messages considered as words over $\Sigma_{\mathcal{N}}$ have always a balanced number of opening parentheses, i.e., "$\mathsf{enc}_a($" and "$\mathsf{hash}($", and closing parentheses, i.e., "$)$".

A message transducer reads a message (interpreted as a word) from left to right, thereby producing some output. If messages are considered as finite trees (where leaves are labeled with atomic messages and internal nodes are labeled with the encryption or hash symbol), a message transducer traverses such a tree from top to bottom and from left to right.

Definition 7 *A* message transducer *$\mathcal{A}$ (over $\mathcal{N}$) is a tuple $(Q, \Sigma_{\mathcal{N}}, I, \Delta, F)$ such that $(Q, \Sigma_{\mathcal{N}}, \Sigma_{\mathcal{N}}, I, \Delta, F)$ is a transducer with letter transitions, and*

1. *for every $x \in \mathcal{M}_\varepsilon$, $\mathcal{A}(x) \subseteq \mathcal{M}_\varepsilon$; and*
2. *for all $p, q \in Q$, $x \in \mathcal{M}$, and $y \in \Sigma_{\mathcal{N}}^*$, if $p(x, y)q \in_s \mathcal{A}$, then $y \in \mathcal{M}_\varepsilon$.*

The first property is a condition on the "external behavior" of a message transducer: Whenever a message transducer gets a message as input, then the corresponding outputs are also messages (rather than arbitrary words). Note that in an attack, the input to a message transducer is always a message. The second property imposes some restriction on the "internal behavior" of a message transducer. Both properties do not seem to be too restrictive. They should be satisfied for most protocols; at least they are for the transducers in the model of the recursive authentication protocol (as described in [14]).

An open issue is whether the properties on the internal and external behavior are decidable, i.e., given a transducer over $\Sigma_{\mathcal{N}}$ does it satisfy 1. and 2. of Definition 7. The main problem is the quantification over messages, i.e., over a context-free rather than a regular set. Nevertheless, in the model of the recursive authentication protocol it is easy to see that the transducers constructed satisfy the properties.

For $S, T \subseteq Q$, we define $M_{\mathcal{A}}(S, T) := \mathcal{A}(S, T) \cap (Q \times \mathcal{M}_\varepsilon \times \Sigma_{\mathcal{N}}^* \times Q)$. By the definition of message transducers, $M_{\mathcal{A}}(I, F) \subseteq (Q \times \mathcal{M}_\varepsilon \times \mathcal{M}_\varepsilon \times Q)$ if I is the set of initial states and F is the set of final states of $\mathcal{A}$. Thus, message transducers specify receive-send actions of principals (in the sense of Definition 1) in a natural way.

In order to define one principal (i.e., the whole sequence of receive-send actions a principal performs) by a single transducer, we consider so-called extended message transducers: $\mathcal{A} = (Q, \Sigma_{\mathcal{N}}, \Delta, (I_0, \dots, I_n))$ is an *extended message-transducer* if $\mathcal{A}_{I_j, I_{j+1}} := (Q, \Sigma_{\mathcal{N}}, I_j, \Delta, I_{j+1})$ is a message transducer for all $j < n$. Given such an extended message transducer, it defines the principal (Q, I_0, n, α) with $\alpha(j) = M_{\mathcal{A}_{I_j, I_{j+1}}}(I_j, I_{j+1})$ for $j < n$. In this setting, an internal action of a principal corresponds to applying one transition in the extended message transducer.

Definition 8 *A* transducer-based protocol *P is a generic protocol where the principals are defined by extended message transducers.*

5.1 Discussion of the Transducer-based Protocol Model

In this section, we aim at clarifying capabilities and limitations of the transducer-based protocol model. To this end, we compare this model with the models usually used for closed-ended protocols. To make the discussion more concrete, we concentrate on the model proposed by Rusinowitch and Turuani (see Section 4), which, among the decidable models used for closed-ended protocols, is very powerful. In what follows, we refer to their model as the *rewriting model*. As pointed out in Section 3, the main difference between the two models is the way receive-send actions are described. In the rewriting model receive-send actions are described by single rewrite rules and in the transducer-based model by message transducers.

Let us start to explain the capabilities of message transducers compared to rewrite rules.

Open-ended data structures. As mentioned in Section 4, with a single rewrite rule one cannot process an

unbounded number of data fields. This is, however, possible with transducers.

For example, considering the IKE protocol (see Section 2), it is easy to specify a transducer which i) reads a list of SAs, each given as a sequence of atomic messages, ii) picks one SA, and iii) returns it. With a single rewrite rule, one could not parse the whole list of SAs.

The transducer-based model of the recursive authentication protocol (described in [14]) shows that transducers can also handle more involved open-ended data structures: The server in this protocol generates a sequence of certificates (containing session keys) from a *request message* of the form $\mathsf{hash}(m_0\mathsf{hash}(m_1 \cdots \mathsf{hash}(m_n)\cdots)$, where the m_i's are sequences of atomic messages and the nesting depth of the hashes is a priori unbounded (see [14] for the exact definition of the messages.)

Of course, a transducer cannot match opening and closing parenthesis, if they are nested arbitrarily deep, since messages are interpreted as words. However, often this is not necessary: In the IKE protocol, the list of SAs is a message without any nesting. In the recursive authentication protocol, the structure of request messages is very simple, and can be parsed by a transducer. Note that a transducer does not need to check whether the number of closing parenthesis in the request message matches the number of hashes because all words sent to a message transducer (by the intruder) are messages, and thus, well-formed.

Simulating rewrite rules. Transducers can simulate certain receive-send actions described by single rewrite rules. Consider for example the rule $\mathsf{enc}_k(x) \rightarrow \mathsf{hash}(kx)$, where x is a variable and k an atomic message: First, the transducer would read "$\mathsf{enc}_k($" and output "$\mathsf{hash}(k$", and then read, letter by letter, the rest of the input message, i.e., "$x)$" – more precisely, the message substituted for x – and simultaneously write it into the output.

Let us now turn to the limitations of the transducer-based model compared to the rewriting model. The main limitations are the following:

1. *Finite memory:* In the rewriting model, principals can store messages of arbitrary size to use them in subsequent receive-send actions. This is not possible with transducers, since they only have finite memory.

2. *Comparing messages:* In the rewriting model, principals can check whether submessages of the input message coincide. For example, if $t = \mathsf{hash}(kx)x$, with k an atomic message and x a variable, a principal can check whether plain text and hash match. Transducers cannot do this.

3. *Copying messages:* In the rewriting model, principals can copy messages of arbitrary size. For example, in the rule $\mathsf{enc}_k(x) \rightarrow \mathsf{hash}(kx)x$, the message x is copied. Again, a transducer would need to store x in some way, which is not possible because of the finite memory. As illustrate above, a transducer could however simulate a rule such as $\mathsf{enc}_k(x) \rightarrow \mathsf{hash}(kx)$.

4. *Linear terms:* A transducer cannot simulate all rewrite rules with linear left and right hand-side. Consider for example the rule $\mathsf{enc}_k(xAy) \rightarrow \mathsf{hash}(yAx)$, where x and y are variables, and A is an atomic message. Since in the output, the order of x and y is switched, a transducer would have to store the messages substituted for x and y. However, this requires unbounded memory.

The undecidability results presented in Section 4 indicate that, if open-ended data structures are involved, the restrictions 1. and 2. seem to be unavoidable. The question is whether this is also the case for the remaining two restrictions. We will comment on this below.

First, let us point out some work-arounds. In 1., it often (at least under reasonable assumptions) suffices to store atomic messages such as principal names, keys, and nonces. Thus, one does not always need unbounded memory. One example is the recursive authentication protocol. In 4., it might be possible to modify the linear terms such that they can be parsed by a message transducer, and such that the security of the protocol is not affected. In the example, if one changes the order of x and y in the output, the rewrite rule can easily be simulated by a transducer. Finally, a work-around for the restrictions 2. to 4., is to put a bound on the size of messages that can be substituted for the variables. This approach is usually pursued in protocol analysis based on finite-state model checking (e.g., [18]), where, however, transducers have the additional advantage of being able to process open-ended data structures. For messages of bounded size, all transformations performed by rewrite rules can also be carried out by message transducers. Moreover, in this setting message transducers can handle type flaws.

Of course, it is desirable to avoid such work-arounds if possible to make the analysis of a protocol more precise and reliable. One approach, which might lift some of the restrictions (e.g., 3. and 4.), is to consider tree transducers instead of word transducers to describe receive-send actions. It seems, however, necessary to devise new kinds of tree transducers or extend existing once, for example tree transducers with look-ahead, that are especially tailored to modeling receive-send actions. A second approach is to combine different computational models for receive-send actions. For instance, a hybrid model in which some receive-actions are described by rewrite rules and others by transducers might still be decidable.

6 The Main Result

The main technical result of this paper is the following:

Theorem 9 *For transducer-based protocols,* ATTACK *is decidable and PSPACE-hard.*

In what follows, we sketch the proof idea of the theorem. See [14] for the detailed proof.

The hardness result is easy to show. It is by reduction from the finite automata intersection problem, which has been shown to be PSPACE-complete by Kozen [11].

The decidability result is much more involved, because we have two sources of infinite behavior in the model. First, the intruder can perform an unbounded number of steps to derive a new message, and second, to perform one receive-send action, a principal can carry out an unbounded number of internal actions. Note that because transducers may have ε-transitions, i.e., not in every transition a letter is read from the input, the number of transitions taken in one receive-send action is not even bounded in the size of the input message or the problem instance.

While the former source of infinity was already present in the (decidable) models for closed-ended protocols [21, 2, 13], the latter is new. To prove Theorem 9, one therefore not only needs to show that the number of actions performed by the intruder can be bounded, but also the number of internal actions of principals. In fact, it suffices to establish the latter, since if we can bound the number of internal actions, a principal only reads messages of bounded length and therefore the intruder only needs to produce messages of size bounded by this length. To bound the number of internal actions, we apply a pumping argument showing that long paths in a message transducer can be truncated. This argument uses that principals (the extended message transducers describing them) have *finite* memory.

More formally, we will show that the following problem is decidable. This immediately implies Theorem 9.

PATHPROBLEM. *Given a finite set $\mathcal{K} \subseteq \mathcal{M}_\varepsilon$ and $k \geq 0$ message transducers $\mathcal{A}_0, \ldots, \mathcal{A}_{k-1}$ with $\mathcal{A}_i = (Q_i, \Sigma_\mathcal{N}, \{q_i^I\}, \Delta_i, \{q_i^F\})$ for $i < k$, decide whether there exist messages $m_i, m_i' \in \mathcal{M}_\varepsilon$, $i < k$, such that*

1. *$m_i \in \mathsf{d}(\mathcal{K} \cup \{m_0', \ldots, m_{i-1}'\})$ for every $i < k$,*
2. *$q_i^I(m_i, m_i')q_i^F \in \mathcal{A}_i$ for every $i < k$, and*
3. *$\mathsf{secret} \in \mathsf{d}(\mathcal{K} \cup \{m_0', \ldots, m_{k-1}'\})$.*

We write an instance of the PATHPROBLEM as $(\mathcal{K}, \mathcal{A}_0, \ldots, \mathcal{A}_{k-1})$ and a solution of such an instance as a tuple $(m_0, m_0', \ldots, m_{k-1}, m_{k-1}')$ of messages. The size of instances is defined as the size of the representation for $\mathcal{K}$ and $\mathcal{A}_0, \ldots, \mathcal{A}_{k-1}$.

Using a pumping argument, we show that in order to find the messages m_i, m_i', for every $i < k$, it suffices to consider paths from q_i^I to q_i^F in $\mathcal{A}_i$ bounded in length by the size of the problem instance – the argument will also show that the bounds can be computed effectively. Thus, a decision procedure can enumerate all paths of length restricted by the (computed) bound and check whether their labels satisfy the conditions. (Note that for every message m and finite set $\mathcal{K}' \subseteq \mathcal{M}_\varepsilon$, $m \in \mathsf{d}(\mathcal{K}')$ can be decided.) In particular, as a "by-product" our decision procedure will yield an actual attack (if any).

The pumping argument. First, we define a *solvability preserving (quasi-)ordering* on messages, which allows to replace single messages in the intruder knowledge by new ones such that if in the original problem a successful attack exists, then also in the modified problem. This reduces the pumping argument to the following problem: Truncate paths in message transducers in such a way that the output of the original path is equivalent (w.r.t. the solvability preserving ordering) to the output of the truncated path. It remains to find criteria for truncating paths in this way. To this end, we introduce another quasi-ordering, the so-called *path truncation ordering*, which indicates at which positions a path can be truncated. To really obtain a bound on the length of paths, it then remains to show that the equivalence relation corresponding to the path truncation ordering has finite index – more accurately, an index that can be bounded in the size of the problem instance. With this, and the fact that message transducers have only finite memory, the length of paths can be restricted. Finally, to show the bound on the index, one needs to establish a bound on the depth of messages (i.e., the depth of nested encryptions and hashes) in successful attacks. Again, we make use of the fact that message transducers have only finite memory.

In what follows, the argument is described in more detail. Due to lack of space, the formal definitions of the orderings as well as the proofs of their properties are omitted. They can be found in the technical report.

Preserving the solvability of instances of the path problem. For every $i \leq k$, we define a quasi-ordering[4] on messages $\preceq_i$ (the so-called *solvability preserving ordering*) which depends on the transducers $A_i, \ldots, A_{k-1}$ and has the following property, which we call (*): For every solvable instance $(\mathcal{K}, \mathcal{A}_i, \ldots, A_{k-1})$ of the path problem, every $m \in \mathcal{K}$, and $\overline{m} \in \mathcal{M}_\varepsilon$ with $m \preceq_i \overline{m}$, the instance $((\mathcal{K} \setminus \{m\}) \cup \{\overline{m}\}, \mathcal{A}_i, \ldots, A_{k-1})$ is solvable as well.

Assume that a path $q_i^I(m_i, m_i')q_i^F \in \mathcal{A}_i$ is replaced by a shorter path such that the corresponding input and output labels of the shorter path, say $\overline{m}_i$ and $\overline{m}_i'$, satisfy $\overline{m}_i \in \mathsf{d}(\mathcal{K} \cup \{m_0', \ldots, m_{i-1}'\})$ and $m_i' \preceq_{i+1} \overline{m}_i'$. Then, after $\mathcal{A}_i$ has returned $\overline{m}_i'$ on input $\overline{m}_i$, the resulting intruder knowledge is $\mathcal{K} \cup \{m_0', \ldots, m_{i-1}', \overline{m}_i'\}$ instead of $\mathcal{K} \cup \{m_0', \ldots, m_{i-1}', m_i'\}$. Using (*), we conclude that there still exists a solution for the rest of the instance, i.e., for $(\mathcal{K} \cup \{m_0', \ldots, m_{i-1}', \overline{m}_i'\}, \mathcal{A}_{i+1}, \ldots, A_{k-1})$.

Consequently, it remains to find criteria for truncating long paths such that

[4] a reflexive and transitive ordering

1. $\overline{m}_i \in \mathsf{d}(\mathcal{K} \cup \{m'_0, \ldots, m'_{i-1}\})$ and
2. $m'_i \preceq_{i+1} \overline{m}'_i$.

Truncating paths such that Condition 1. is satisfied is rather easy. The involved part is Condition 2. To this end, we introduce the path truncation ordering.

Truncating paths. We extend $\preceq_i$ to a quasi-ordering $\preceq_i^l$ (the *path truncation ordering*) on so-called left half-messages. *Left half-messages* are prefixes of messages (considered as words over $\Sigma_{\mathcal{N}}$). In particular, left half-messages may lack some closing parentheses. The "l" in $\preceq_i^l$ is the number of missing parentheses (the *level* of left half-messages); $\preceq_i^l$ only relates left half-messages of level l. Analogously, right half-messages are suffixes of messages. Thus, they may have too many closing parentheses; the number of additional parentheses determines the level of right half-messages. The equivalence relation $\equiv_i^l$ on left half-messages corresponding to $\preceq_i^l$ has the following property, which we call (**): For all left half-messages α, α' of level l and right half-messages γ of level l, $\alpha \equiv_i^l \alpha'$ implies $\alpha\gamma \equiv_i \alpha'\gamma$. (Note that $\alpha\gamma$ and $\alpha'\gamma$ are messages.)

Now, consider two positions $x < y$ in the path $\pi = (q_i^I, m_i, m'_i, q_i^F) \in \mathcal{A}_i$ such that α_x, α_y are the output labels up to these positions, and γ_x, γ_y are the output labels beginning at these positions, i.e., $m'_i = \alpha_x\gamma_x = \alpha_y\gamma_y$. Clearly, α_x, α_y are left half-messages and γ_x, γ_y are right half-messages. Assume that α_x, α_y have the same level l (in particular, γ_x, γ_y have level l) and $\alpha_x \equiv_i^l \alpha_y$. Then, by (**) it follows $m'_i = \alpha_y\gamma_y \equiv_i \alpha_x\gamma_y =: \overline{m}'_i$, where $\overline{m}'_i$ is the output label of the path obtained by cutting out the subpath in π between x and y.[5] Thus, $\equiv_i^l$ provides us with the desired criterion for "safely" (in the sense of Condition 2.) truncating paths. In order to conclude that the length of paths can be bounded in the size of the problem instance, it remains to show that l and the index of $\equiv_i^l$ (i.e., the number of equivalence classes modulo $\equiv_i^l$ on left half-messages of level l) can be bounded in the size of the problem instance. To this end, the following is shown.

Bounding the depth of messages. We first show (***): If $(m_0, m'_0, \ldots, m_{k-1}, m'_{k-1})$ is a solution of $(\mathcal{K}, \mathcal{A}_0, \ldots, \mathcal{A}_{k-1})$, then, for every i, their also exists a solution if the depth of m_i is bounded in the size of the problem instance.

We then show how the depth of the output message m'_i can be bounded: Let π be a path in $\mathcal{A}_i$ from q_i^I to q_i^F or a strict path in $\mathcal{A}_i$, and x be a position in π such that α_x is the input label of π up to position x and β_x is the output label of π up to x. Then, the level of β_x can be bounded by a polynomial in the level of α_x and the number of states of $\mathcal{A}_i$. As a corollary, one obtains that the depth of output messages can be bounded in the depth of input messages, and using (***), that both the depth of input and output messages can be bounded in the size of the problem instance.

With this, one can show that the index of $\equiv_i^l$ is bounded. Moreover, the l in 2. (the level of the half-messages α_x, α_y, γ_x, γ_y) is bounded in the size of the problem instance. Therefore, $\equiv_i^l$ can serve as the desired criterion for truncating paths.

[5] One little technical problem is that $\alpha_x\gamma_y$ does not need to be a message since it may contain a word of the form $\mathsf{enc}_a()$, which is not a message. However, if one considers three positions $x < y < z$, then one can show that either $\alpha_x\gamma_y$ or $\alpha_y\gamma_z$ is a message.

7 Conclusion

We have introduced a generic protocol model for analyzing the security of open-ended protocols, i.e., protocols with open-ended data structures, and investigated the decidability of different instances of this model. In one instance, receive-send actions are modeled by sets of rewrite rules. We have shown that in this instance, security is undecidable. This result indicated that to obtain decidability, principals should only have finite memory and should not be able to compare messages of arbitrary size. This motivated our transducer-based model, which complies to these restrictions, but still captures certain open-ended protocols. We have shown that in this model security is decidable and PSPACE-hard; it remains to establish a tight complexity bound. These results have been shown for the shared key setting and secrecy properties. We conjecture that they carry over rather easily to public key encryption and authentication.

As pointed out in Section 5.1, a promising future direction is to combine the transducer-based model with the models for closed-ended protocols and to devise tree transducers suitable for describing receive-send actions. We will also try to incorporate complex keys, since they are used in many protocols. We believe that the proof techniques devised in this paper will help to show decidability also in the more powerful models. Finally, encouraged by the work that has been done for closed-ended protocols, the long-term goal of the work started here is to develop tools for automatic verification of open-ended protocols, if possible by integrating the new algorithms into existing tools.

Acknowledgement I would like to thank Thomas Wilke and the anonymous referees for useful comments and suggestions for improving the presentation of this paper. Thanks also to Catherine Meadows for pointing me to the paper by Pereira and Quisquater.

Bibliography

[1] R.M. Amadio and W. Charatonik. On name generation and set-based analysis in Dolev-Yao model.

Technical Report RR-4379, INRIA, 2002.

[2] R.M. Amadio, D. Lugiez, and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. Technical Report RR-4147, INRIA, 2001.

[3] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *Proceedings of the 5th ACM Conference on Computer and Communication Secruity (CCS'98)*, pages 17–26, San Francisco, CA, 1998. ACM Press.

[4] J. Bryans and S.A. Schneider. CSP, PVS, and a Recursive Authentication Protocol. In *DIMACS Workshop on Formal Verification of Security Protocols*, 1997.

[5] J.A. Bull and D.J. Otway. The authentication protocol. Technical Report DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03, Defence Research Agency, Malvern, UK, 1997.

[6] D. Dolev, S. Even, and R.M. Karp. On the Security of Ping-Pong Protocols. *Information and Control*, 55:57–68, 1982.

[7] D. Dolev and A.C. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[8] N.A. Durgin, P.D. Lincoln, J.C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[9] S. Even and O. Goldreich. On the Security of Multi-Party Ping-Pong Protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pages 34–39, 1983.

[10] N. Ferguson and B. Schneier. A Cryptographic Evaluation of IPsec. Technical report, 2000. Available from http://www.counterpane.com/ipsec.pdf.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[12] D. Harkins and D. Carrel. *The Internet Key Exchange (IKE)*, November 1998. RFC 2409.

[13] A. Huima. Efficient infinite-state analysis of security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[14] R. Küsters. On the Decidability of Cryptographic Protocols with Open-ended Data Structures. Technical Report 0204, Institut für Informatik und Praktische Mathematik, CAU Kiel, Germany, 2002. Available from http://www.informatik.uni-kiel.de/reports/2002/0204.html.

[15] G. Lowe. Towards a Completeness Result for Model Checking of Security Protocols. *Journal of Computer Security*, 7(2–3):89–146, 1999.

[16] C. Meadows. Extending formal cryptographic protocol analysis techniques for group protocols and low-level cryptographic primitives. In P. Degano, editor, *Proceedings of the First Workshop on Issues in the Theory of Security (WITS'00)*, pages 87–92, 2000.

[17] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of DISCEX 2000*, pages 237–250. IEEE Computer Society Press, 2000.

[18] J. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols using Murphi. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, 1997.

[19] L.C. Pauslon. Mechanized Proofs for a Recursive Authentication Protocol. In *10th IEEE Computer Security Foundations Workshop (CSFW-10)*, pages 84–95, 1997.

[20] O. Pereira and J.-J. Quisquater. A Security Analysis of the Cliques Protocols Suites. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 73–81, 2001.

[21] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 174–190, 2001.

[22] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to key agreement. In *IEEE International Conference on Distributed Computing Systems*, pages 380–387. IEEE Computer Society Press, 1998.

[23] S. D. Stoller. A bound on attacks on authentication protocols. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science*. Kluwer, 2002. To appear.

[24] J. Zhou. Fixing a security flaw in IKE protocols. *Electronic Letter*, 35(13):1072–1073, 1999.

Game Strategies In Network Security

Kong-Wei Lye
Department of Electrical and Computer Engineering
kwlye@cmu.edu

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3890, USA
wing@cs.cmu.edu

Abstract

This paper presents a game-theoretic method for analyzing the security of computer networks. We view the interactions between an attacker and the administrator as a two-player stochastic game and construct a model for the game. Using a non-linear program, we compute Nash equilibria or best-response strategies for the players (attacker and administrator). We then explain why the strategies are realistic and how administrators can use these results to enhance the security of their network.

Keywords: Stochastic Games, Non-linear Programming, Network Security.

1 Introduction

Government agencies, schools, retailers, banks, and a growing number of goods and service providers today all use the Internet as their integral way of conducting daily business. Individuals, good or bad, can also easily connect to the internet. Due to the ubiquity of the Internet, computer security has now become more important than ever to organizations such as governments, banks, and businesses. Security specialists have long been interested in knowing what an intruder can do to a computer network, and what can be done to prevent or counteract attacks. In this paper, we describe how game theory can be used to find strategies for both an attacker and the administrator. We illustrate our approach in an example (Figure 1) of a local network connected to the Internet and consider the interactions between them as a general-sum stochastic game. In Section 2, we introduce the formal model for stochastic games and relate the elements of this model to those in our network example. In Section 3, we explain the concept of a Nash equilibrium for stochastic games and explain what it means to the attacker and administrator. Then, in Section 4, we describe three possible attack scenarios for our network example. In these scenarios, an attacker on the Internet attempts to deface the homepage on the public web server on the network, launch an internal denial-of-service (DOS) attack, and capture some important data from a workstation on the network. We compute Nash equilibria (best responses) for the attacker and administrator using a non-linear program and explain one of the solutions found for our example in Section 5. We discuss the implications of our approach in Section 6 and compare our work with previous work in the literature in Section 7. Finally, we summarize our results and point to future directions in Section 8.

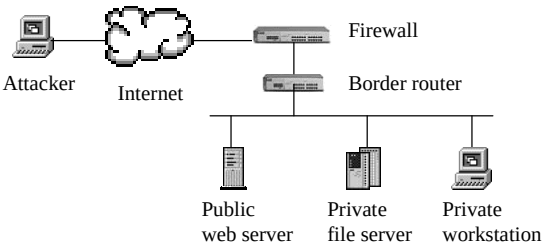


Figure 1: A Network Example

2 Networks as Stochastic Games

In this section, we first introduce the formal model of a *stochastic game*. We then use this model for our network attack example and explain how the state set, actions sets, cost/reward functions, and transition probabilities can be defined or derived. Formally, a two-player stochastic game is a tuple $(S, A^1, A^2, Q, R^1, R^2, \beta)$ where $S = \{\xi_1, \cdots, \xi_N\}$ is the state set and $A^k = \{\alpha_1^k, \cdots, \alpha_{M^k}^k\}$, $k = 1, 2$, $M^k = |A^k|$, is the action set of player k. The action set for player k at state s is a subset of A^k, i.e., $A_s^k \subseteq A^k$ and $\bigcup_{i=1}^{N} A_{\xi_i}^k = A^k$. $Q : S \times A^1 \times A^2 \times S \to [0, 1]$ is the state transition function. $R^k : S \times A^1 \times A^2 \to \Re$, $k = 1, 2$ is the reward function [1] of player k. $0 < \beta \leq 1$ is a *discount factor* for discounting future rewards, i.e., at the current state, a state transition has a reward worth its full

[1] We use the term "reward" in general here; in later sections, positive values are rewards and negative values are costs.

value, but the reward for the transition from the next state is worth β times its value at the current state.

The game is played as follows: at a discrete time instant t, the game is in state $s_t \in S$. Player 1 chooses an action a_t^1 from A^1 and player 2 chooses an action a_t^2 from A^2. Player 1 then receives a reward $r_t^1 = R^1(s_t, a_t^1, a_t^2)$ and player 2 receives a reward $r_t^2 = R^2(s_t, a_t^1, a_t^2)$. The game then moves to a new state s_{t+1} with conditional probability Prob$(s_{t+1}|s_t, a_t^1, a_t^2)$ equal to $Q(s_t, a_t^1, a_t^2, s_{t+1})$.

In our example, we let the attacker be player 1 and the administrator be player 2. We provide two views of the game: the attacker's view (Figure 3) and the administrator's view (Figure 4). We describe these figures in detail later in Section 4.

2.1 Network state

In general, the state of the network can contain various kinds of features such as type of hardware, software, connectivity, user privileges, etc. Using more features in the state allows us to represent the network better, but often makes the analysis more complex and difficult. We view the network example as a graph (Figure 2). A node in the graph is a physical entity such as a workstation or router. We model the external world as a single computer (node *E*) and represent the web server, file server, and workstation by nodes *W, F,* and *N,* respectively. An edge in the graph represents a direct communication path (physical or virtual). For example, the external computer (node *E*) has direct access to only the public web server (node *W*).

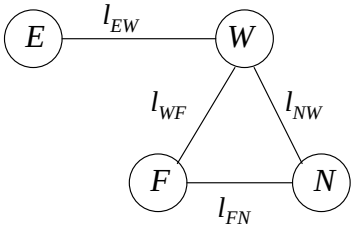


Figure 2: Network State

Instantiating our game model, we let a superstate $< n_W, n_F, n_N, t > \in S$ be the state of the network. n_W, n_F, and n_N are the *node states* for the web server, file server, and workstation respectively, and t is the *traffic state* for the whole network. Each node X (where $X \in \{E, W, F, N\}$) has a node state $n_X =< P, a, d >$ to represent information about hardware and software configurations. $P \subseteq \{f, h, n, p, s, v\}$ is a list of software applications running on the node and f, h, n, and p denote *ftpd, httpd, nfsd,* and some user process respectively. For malicious codes, s and v represent sniffer programs and viruses respectively. $a \in \{u, c\}$ is a variable used to represent the state of the user accounts. u means no user account has been compromised and c means at least one user account has been compromised. We use the variable $d \in \{c, i\}$ to represent the state of the data on the node. c and i mean the data has and has not been corrupted or stolen respectively. For example, if $n_W =< (f, h, s), c, i >$, then the web server is running *ftpd* and *httpd,* a sniffer program has been implanted, and a user account has been compromised but no data has yet been corrupted or stolen.

The traffic information for the whole network is captured in the traffic state $t =< \{l_{XY}\} >$ where X and Y are nodes and $l_{XY} \in \{0, \frac{1}{3}, \frac{2}{3}, 1\}$ indicates the load carried on this link. A value of 1 indicates maximum capacity. For example, in a 10Base-T connection, the values 0, $\frac{1}{3}$, $\frac{2}{3}$, and 1 represent 0Mbps, 3.3Mbps, 6.7Mbps, and 10Mbps respectively. In our example, the traffic state is $t =< l_{EW}, l_{WF}, l_{FN}, l_{NW} >$. We let $t =< \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3} >$ for normal traffic conditions.

The potential state space for our network example is very large but we shall discuss how to handle this problem in Section 6. The full state space in our example has a size of $|n_W| \times |n_F| \times |n_N| \times |t| = (63 \times 2 \times 2)^3 \times 4^4 \approx$ 4 billion states but there are only 18 states (15 in Figure 3 and 3 others in Figure 4) relevant to our illustration here. In these figures, each state is represented using a box with a symbolic state name and the values of the state variables. For convenience, we shall mostly refer to the states using their symbolic state names.

2.2 Actions

An action pair (one from the attacker and one from the administrator) causes the system to move from one state to another in a probabilistic manner. A single action for the attacker can be any part of his attack strategy, such as flooding a server with *SYN* packets or downloading the password file. When a player does nothing, we denote this inaction as ϕ. The action set for the attacker $A^{Attacker}$ consists of all the actions he can take in all the states, $A^{Attacker}$ = { *Attack_httpd, Attack_ftpd, Continue_hacking, Deface_website_leave, Install_sniffer, Run_DOS_virus, Crack_file_server_root_password, Crack_workstation_root_password, Capture_data, Shutdown_network,* ϕ}, where ϕ denotes inaction. His actions in each state is a subset of $A^{Attacker}$. For example, in the state **Normal_operation** (see Figure 3, topmost state), the attacker has an action set $A^{Attacker}_{\textbf{Normal\_operation}}$ = { *Attack_httpd, Attack_ftpd,* ϕ}. Actions for the administrator are mainly preventive or restorative measures. In our example, the administrator has an action set $A^{Administrator}$ = { *Remove_compromised_account_restart_httpd, Restore_website_remove_compromised_account, Remove_virus_compromised_account, Install_sniffer_detector, Remove_sniffer_detector, Remove_compromised_account_restart_ftpd,*

Remove_compromised_account_sniffer, $\phi\}$. In state **Ftpd_attacked** (see Figure 4), the administrator has an action set $A^{Adminstrator}_{\mathbf{Ftpd\_attacked}} = \{$ *install_sniffer_detector*, $\phi\}$.

A node with a compromised account may or may not be observable by the administrator. When it is not observable, we model the situation as the administrator having an empty action set in the state. We assume that the administrator does not know whether there is an attacker or not. Also, the attacker may have several objectives and strategies that the administrator does not know. Furthermore, not all of the attacker's actions can be observed.

2.3 State transition probabilities

In our example, we assign state transition probabilities based on intuition. In real life, case studies, statistics, simulations, and knowledge engineering can provide the required probabilities. In Figures 3 and 4, state transitions are represented by arrows. Each arrow is labeled with an action, a transition probability, and a cost/reward. In the formal game model, a state transition probability is a function of both players' actions. Such probabilities are used in the non-linear program (Section 3) for computing a solution to the game. However, in order to separate the game into two views, we show the transitions as simply due to a single player's actions. For example, with the second dashed arrow from the top in Figure 3, we show the derived probability Prob(**Ftpd_hacked**|**Ftpd_attacked**, *Continue_attacking*) = 0.5 as due to only the attacker's action *Continue_attacking*. When the network is in state **Normal_operation** and neither the attacker nor administrator takes any action, it will tend to stay in the same state. We model this situation as having a near-identity stochastic matrix, i.e., we let Prob(**Normal_operation**| **Normal_operation**, ϕ, ϕ)=1-ϵ for some small $\epsilon < 0.5$. Then Prob(s| **Normal_operation**, ϕ, ϕ)=$\frac{\epsilon}{N-1}$ for all $s \neq$ **Normal_operation** where N is the number of states. There are also state transitions that are infeasible. For example, it may not be possible for the network to move from a normal operation state to a completely shutdown state without going through some intermediate states. Infeasible state transitions are assigned transition probabilities of 0.

2.4 Costs and rewards

There are costs (negative values) and rewards (positive values) associated with the actions of the administrator and attacker. The attacker's actions have mostly rewards and such rewards are in terms of the amount of damage he does to the network. Some costs, however, are difficult to quantify. For example, the loss of marketing strategy information to a competitor can cause large monetary losses. A defaced corporate website may cause the company to lose its reputation and its customers to lose confidence. Meadows's work on cost-based analysis of DOS discusses how costs can be assigned to an attacker's actions using categories such as *cheap*, *medium*, *expensive*, and *very expensive* [Mea01].

In our model, we restrict ourselves to the amount of recovery effort (time) required by the administrator. The reward for an attacker's action is mostly defined in terms of the amount of effort the administrator has to make to bring the network from one state to another. For example, when a particular service crashes, it may take the administrator 10 or 15 minutes of time to determine the cause and restart the service [2]. In Figure 4, it costs the administrator 10 minutes to remove a compromised user account and to restart the *httpd* (from state **Httpd_hacked** to state **Normal_operation**). For the attacker, this amount of time would be his reward. To reflect the severity of the loss of the important financial data in our network example, we assign a very high reward for the attacker's action that leads to the state where he gains this data. For example, from state **Workstation_hacked** to state **Workstation_data_stolen_1** in Figure 3, the reward is 999. There are also some transitions in which the cost to the administrator is not the same magnitude as the reward to the attacker. It is such transitions that make the game a general-sum game instead of a zero-sum game.

3 Nash Equilibrium

We now return to the formal model for stochastic games. Let $\Omega^n = \{p \in \Re^n \mid \sum_{i=1}^n p_i = 1, p_i \geq 0\}$ be the set of probability vectors of length n. $\pi^k : S \rightarrow \Omega^{M^k}$ is a stationary strategy for player k. $\pi^k(s)$ is the vector $[\pi^k(s,\alpha_1) \quad \cdots \quad \pi^k(s,\alpha_{M^k})]^\mathsf{T}$ where $\pi^k(s,\alpha)$ is the probability that player k should use to take action α in state s. A stationary strategy π^k is a strategy which is independent of time and history. A mixed or randomized stationary strategy is one where $\pi^k(s,\alpha) \geq 0 \; \forall s \in S$ and $\forall \alpha \in A^k$ and a pure strategy is one where $\pi^k(s,\alpha_i) = 1$ for some $\alpha_i \in A^k$.

The objective of each player is to maximize some expected return. Let s_t be the state at time t and r^k_t be the reward received by player k at time t. We define an expected return to be the column vector $v^k_{\pi^1,\pi^2} = [v^k_{\pi^1,\pi^2}(\xi_1) \quad \cdots \quad v^k_{\pi^1,\pi^2}(\xi_N)]^\mathsf{T}$ where $v^k_{\pi^1,\pi^2}(s) = E_{\pi^1,\pi^2}\{\sum_{n=0}^N (\beta)^n r^k_{t+n} \mid s_t = s\}$. The expectation operator $E_{\pi^1,\pi^2}\{\cdot\}$ is used to mean that player k plays π^k, i.e., player k chooses an action using the probability distribution $\pi^k(s_{t+n})$ at s_{t+n} and receives an immediate reward $r^k_{t+n} = \pi^1(s_{t+n})^\mathsf{T} R^k(s_{t+n})\pi^2(s_{t+n})$ for $n \geq 0$. $R^k(s) = [R^k(s,a^1,a^2)]_{a^1 \in A^1, a^2 \in A^2}$ [3], $k = 1, 2$ is

[2] These numbers were given by the department's network manager.

[3] We use $[m(i,j)]_{i \in I, j \in J}$ to refer to an $|I| \times |J|$ matrix with elements $m(i,j)$.

player k's reward matrix in state s.

For an infinite-horizon game, we let $T = \infty$ and use a discount factor $\beta < 1$ to discount future rewards. $v^k(s)$ is then the expected total discounted rewards that player k will receive when starting at state s. For a finite-horizon game, $0 < T < \infty$ and $\beta = 1$. v^k is also called the *value vector* of player k.

A Nash equilibrium in stationary strategies (π^1_*, π^2_*) is one which satisfies $v^1(\pi^1_*, \pi^2_*) \geq v^1(\pi^1, \pi^2_*)\ \forall \pi^1 \in \Omega^{M^1}$ and $v^2(\pi^1_*, \pi^2_*) \geq v^2(\pi^1_*, \pi^2)\ \forall \pi^2 \in \Omega^{M^2}$ component-wise. Here, $v^k(\pi^1, \pi^2)$ is the value vector of the game for player k when both players play their stationary strategies π^1 and π^2 respectively and $\geq$ is used to mean the left-hand-side vector is component-wise, greater than or equal to the right-hand-side vector. At this equilibrium, there is no mutual incentive for either one of the players to deviate from their equilibrium strategies π^1_* and π^2_*. A deviation will mean that one or both of them will have lower expected returns, i.e., $v^1(\pi^1, \pi^2)$ and/or $v^2(\pi^1, \pi^2)$. A pair of Nash equilibrium strategies is also known as best responses, i.e., if player 1 plays π^1_*, player 2's best response is π^2_* and vice versa.

In our network example, π^1 and π^2 corresponds to the attacker's and administrator's strategies respectively. $v^1(\pi^1, \pi^2)$ corresponds to the expected return for the attacker and $v^2(\pi^1, \pi^2)$ corresponds to the expected return for the administrator when they use the strategies π^1 and π^2. In a Nash equilibrium, when the attacker and administrator use their best-response strategies π^1_* and π^2_* respectively, neither will gain a higher expected return if the other continues using his Nash strategy.

Every general-sum discounted stochastic game has at least one Nash equilibrium in stationary mixed strategies (see [FV96]) (not necessarily unique) and finding these equilibria is non-trivial. In our network example, finding multiple Nash equilibria means finding multiple pairs of Nash strategies. In each pair, a strategy for one player is a best-response to the strategy for the other player and vice versa. A non-linear program found in [FV96] can be used to find the equilibrium strategies for both players in a general-sum stochastic game. We shall refer to this non-linear program as NLP-1 and use it to find Nash equilibria for our network example later in Section 5.

4 Attack and Response Scenarios

In this section, we describe three different attack and response scenarios. We show in Figure 3, the viewpoint of the attacker, how he sees the state of the network change as a result of his actions. The viewpoint of the administrator is shown in Figure 4. In both figures, a state is represented using a box containing the symbolic name and the values of the state variables for that state. Each transition is labeled with an action, the probability of the transition, and the gain or cost in minutes of restorative effort incurred on the administrator. The three scenarios are indicated using bold, dotted, and dashed arrows in Figure 3. Due to space constraints, not all state transitions for every action are shown. From one state to the next, state variable changes are highlighted using boldface.

Scenario 1: A common target for use as a launching base in an attack is the public web server. The web server typically runs an *httpd* and an *ftpd* and a common technique for the attacker to gain a root shell is *buffer overflow*. Once the attacker gets a root shell, he can deface the website and leave. This scenario is shown by the state transitions indicated by bold arrows in Figure 3.

From state **Normal_operation**, the attacker takes action *Attack_httpd*. With a probability of 1.0 and a reward of 10, he moves the system to state **Httpd_attacked**. This state indicates increased traffic between the external computer and the web server as a result of his attack action. Taking action *Continue_attacking*, he has a 0.5 probability of success of gaining a user or root access through bringing down the *httpd*, and the system moves to state **Httpd_hacked**. Once he has root access in the web server, he can deface the website, restart the *httpd* and leaves, moving the network to state **Website_defaced**.

Scenario 2: The other thing that the attacker can do after he has hacked into the web server is to launch a DOS attack from inside the network. This is shown by the state transitions drawn using dotted arrows (starts from the middle of Figure 3), with each state having more internal traffic than the previous.

From state **Webserver_sniffer**, the attacker takes action *Run_DOS_virus*. With probability 1 and a reward of 30, the network moves into state **Webserver_DOS_1**. In this state, the traffic load on all internal links has increased from $\frac{1}{3}$ to $\frac{2}{3}$. From this state, the network degrades to state **Webserver_DOS_2** with probability 0.8 even when the attacker does nothing. The traffic load is now at full capacity of 1 in all the links. We assume that there is a 0.2 probability that the administrator notices this and takes action to recover the system. In the very last state, the network grinds to a halt and nothing productive can take place.

Scenario 3: Once the attacker has hacked into the web server, he can install a sniffer and a backdoor program. The sniffer will sniff out passwords from the users in the workstation when they access the file server or web server. Using the backdoor program, the attacker then comes back to collect his password list from the sniffer program, cracks the root password, logs on to the workstation, and searches the local hard disk. This scenario is shown by the state transitions indicated by dashed arrows in Figure 3.

From state **Normal_operation**, the attacker takes action *Attack_ftpd*. With a probability of 1.0 and a reward of 10, he uses the buffer overflow or a similar attack technique and moves the system to state **Ftpd_attacked**. There is increased traffic between the external computer and the web

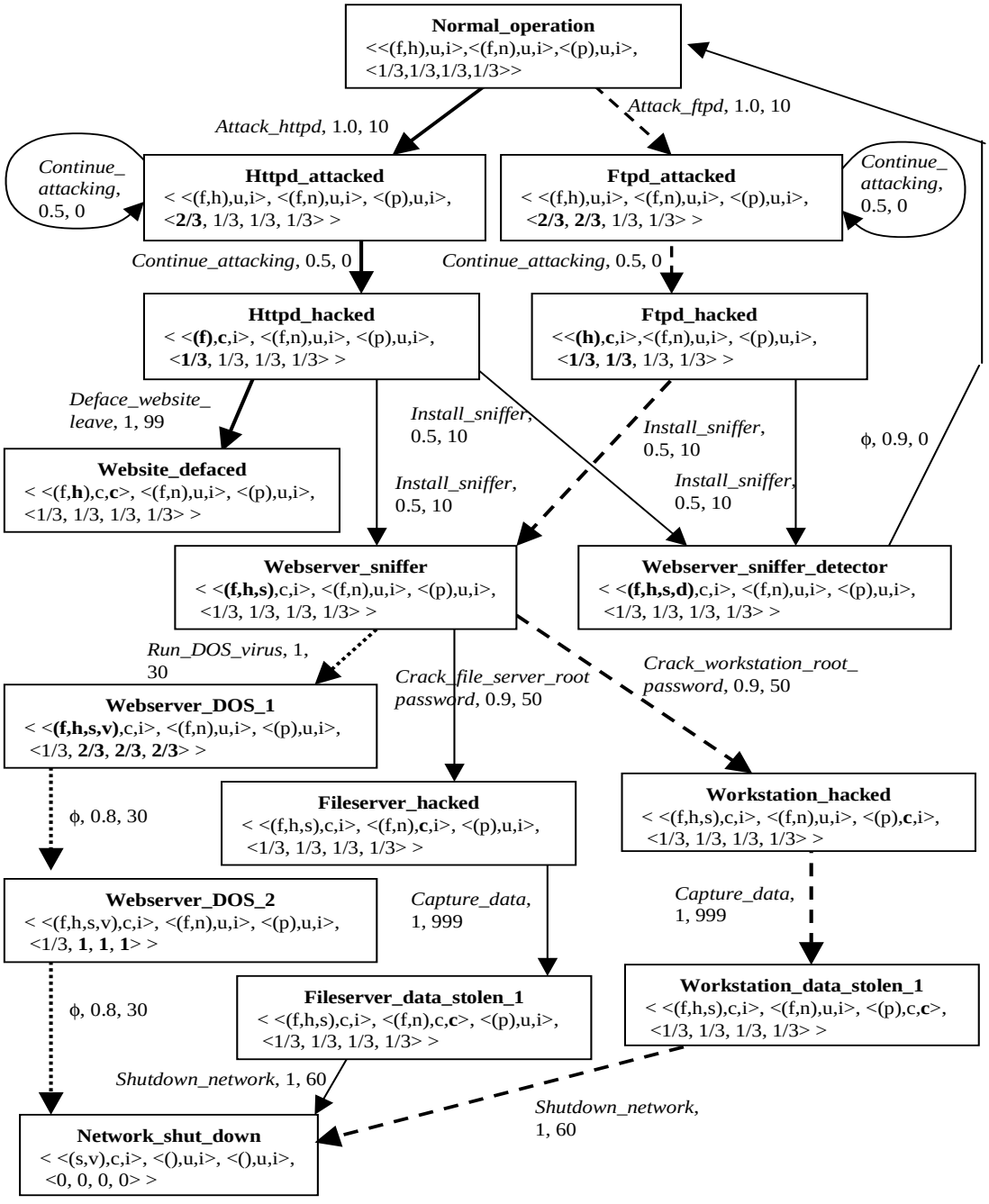


Figure 3: Attacker's view of the game

server as well as between the web server and the file server in this state, both loads going from $\frac{1}{3}$ to $\frac{2}{3}$. If he continues to attack the *ftpd*, he has a 0.5 probability of success of gaining a user or root access through bringing down the *ftpd*, and the system moves to state **Ftpd_hacked**. From here, he can install a sniffer program and with probability 0.5 and a reward of 10, move the system to state **Webserver_sniffer**. In this state, he has also restarted the *ftpd* to avoid causing suspicion from normal users and the administrator. The attacker then collects the password list and cracks the root password on the workstation. We assume he has a 0.9 chance of success and when he succeeds, he gains a reward of 50 and moves the network to state **Workstation_hacked**. To cause more damage to the network, he can even shut it down using the privileges of root user in this workstation.

We now turn our attention to the administrator's view (see Figure 4). The administrator in our example does mainly restorative work and his actions can be restarting the *ftpd*, removing a virus, etc. He also takes preventive measures and such actions can be installing a sniffer detector, re-configuring a firewall, deactivating a user account, and so on. In the first attack scenario in which the attacker defaces the website, the administrator can only take the action *Restore_website_remove_compromised_account* to bring the network from state **Website_defaced** to **Normal_operation**. In the second attack scenario, the states **Webserver_DOS_1** and **Webserver_DOS_2** (indicated by double boxes) show the network suffering from the effects of the internal DOS attack. All the administrator can do is take the action *Remove_virus_compromised_account* to bring the network back to **Normal_operation**. In the third attack scenario, there is nothing he can do to restore the network back to its original operating state. Important data has been stolen and no action allows him to undo this situation. The network can only move from state **Workstation_data_stolen_1** to **Workstation_data_stolen_2** (indicated by dotted box on bottom right in Figure 4).

The state **Ftpd_attacked** (dashed box) is an interesting state because here, the attacker and administrator can engage in real-time game play. In this state, when the administrator notices an unusual increase in traffic between the external network and the web server and also between the web server and the file server, he may suspect an attack is going on and take action *Install_sniffer_detector*. Taking this action, however, incurs a cost of 10. If the attacker is still attacking, the system moves into state **Ftpd_attacked_detector**. If he has already hacked into the web server, then the system moves to state **Webserver_sniffer_detector**. Detecting the sniffer program, the administrator can now remove the affected user account and the sniffer program to prevent the attacker from further attack actions.

5 Results

We implemented NLP-1 (non-linear program mentioned in Section 3) in *MATLAB*, a mathematical computation software package by *The MathWorks, Inc.* To run NLP-1, the cost/reward and state transition functions defined in Section 2 are required. In the formal game model, the state of the game evolves only at discrete time instants. In our example, we imagine that the players take actions only at discrete time instants. The game model also requires actions to be taken simultaneously by both players. There are some states in which a player has only one or two non-trivial actions and for consistency and easier computation using NLP-1, we add an *inaction* ϕ to the action set for the state so that the action sets are all of the same cardinality. Overall, our game model has 18 states and 3 actions per state.

We ran NLP-1 on a computer equipped with a 600Mhz Pentium-III and 128Mb of RAM. The result of one run of NLP-1 is a Nash equilibrium. It consists of a pair of strategies ($\pi_*^{Attacker}$ and $\pi_*^{Administrator}$) and a pair of value vectors ($v_*^{Attacker}$ and $v_*^{Administrator}$) for the attacker and administrator. The strategy for a player consists of a probability distribution over the action set for each state and the value vector consists of a state value for each state. We ran NLP-1 on 12 different sets of initial conditions and found 3 different Nash equilibria. Each run took 30 to 45 minutes. Due to space constraints, however, we shall discuss only one, shown in Table 1.

We explain the strategies for some of the more interesting states here. For example, in the state **Httpd_hacked** (5^{th} row in Table 1), the attacker has action set $\{Deface\_website\_leave, Install\_sniffer, \phi\}$. His strategy for this state says that he should use *Deface_website_leave* with probability 0.33 and *Install_sniffer* with probability 0.10. Ignoring the last action ϕ, and after normalizing, these probabilities become 0.77 and 0.23 respectively for *Deface_website_leave* and *Install_sniffer*. Even though installing a sniffer may allow him to crack a root password and eventually capture the data he wants, there is also the possibility that the system administrator detects his presence and takes preventive measures. He is thus able to do more damage (probabilistically speaking) if he simple defaces the website and leaves. In this same state, the administrator can either take action *Remove_compromised_account_restart_httpd* or action *Install_sniffer_detector*. His strategy says that he should take the former with probability 0.67 and the latter with probability 0.19. Ignoring the third action ϕ and after normalizing, these probabilities become 0.78 and 0.22 respectively. This tells him that he should immediately remove the compromised account and restart the *httpd* rather than continue to "play" with the attacker. It is not shown here in our model but installing the sniffer detector could be a step towards apprehending the attacker,

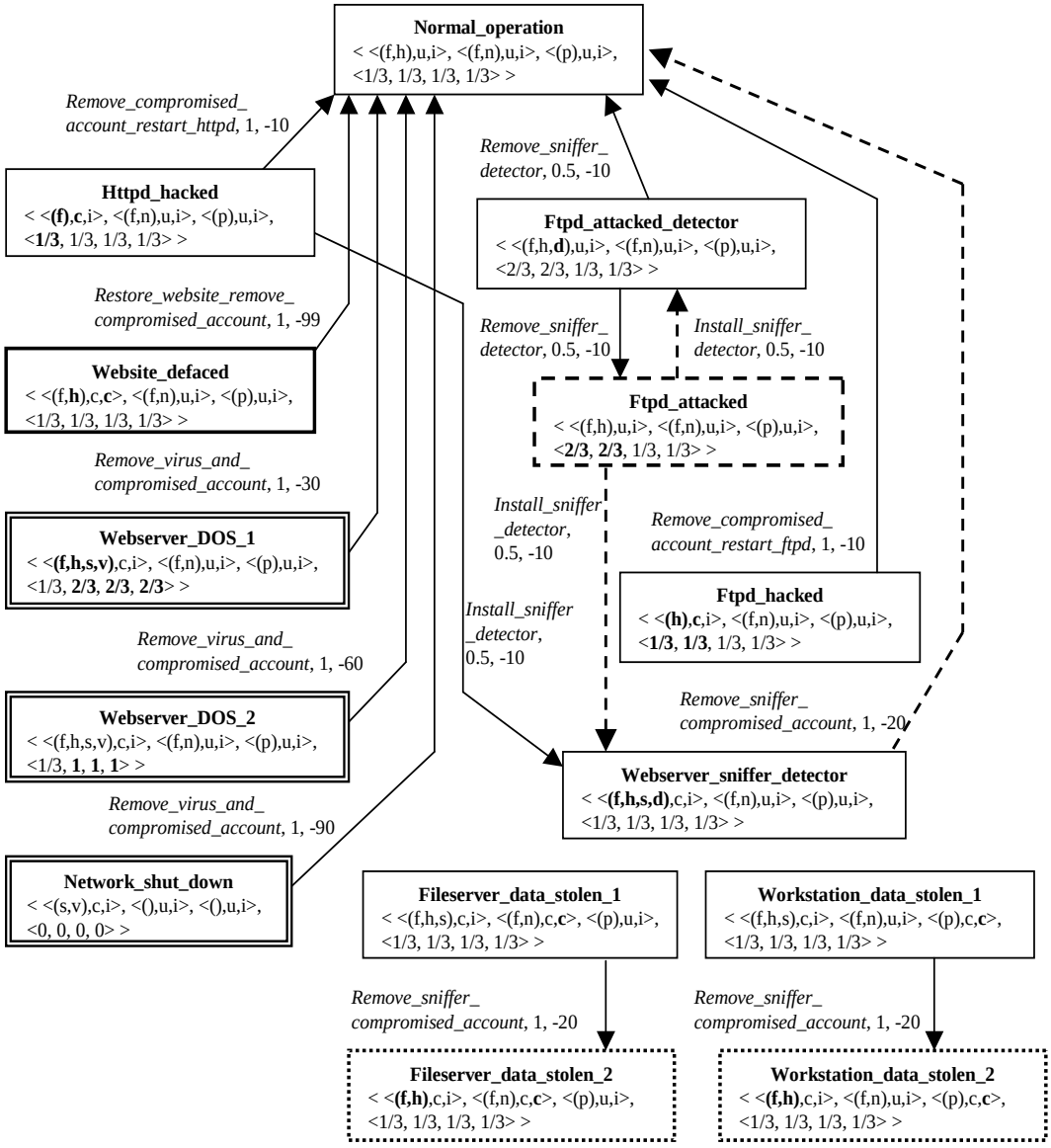


Figure 4: Administrator's view of the game

which means greater reward for the administrator. In the state **Webserver_sniffer** (8^{th} row in Table 1), the attacker should take the actions *Crack_file_server_root_password* and *Crack_workstation_root_password* with equal probability (0.5) because either action will let him do the same amount of damage eventually. Finally, in the state **Webserver_DOS_1** (10^{th} row in Table 1), the system administrator should remove the DoS virus and compromised account, this being his only action in this state (the other two being ϕ).

In Table 1, we note that the value vector for the administrator is not exactly the negative of that for the attacker, that is, in our example, not all state transitions have costs whose corresponding rewards are of the same magnitude. In a zero-sum game, the value vector for one player is the negative of the other's. In this table, the negative state values for the administrator correspond to his expected costs or expected amount of recovery time (in minutes) required to bring the network back to normal operation. Positive state values for the attacker correspond to his expected reward or the expected amount of damage he causes to the administrator (again, in minutes of recovery time). Both the attacker and administrator would want to maximize the state values for all the states.

In state **Fileserver_hacked** (13^{th} row in Table 1), the attacker has gained access into the file server and has full control over the data in it. In state **Workstation_hacked** (15^{th} row in Table 1), the attacker has gained root access to the workstation. These two states have the same value of 1065.5, the highest among all states, because these are

		Strategies		State Values	
	State	Attacker	Administrator	Attacker	Administrator
1	**Normal_operation**	[1.00 0.00 0.00]	[0.33 0.33 0.33]	210.2	-206.8
2	**Httpd_attacked**	[1.00 0.00 0.00]	[0.33 0.33 0.33]	202.2	-191.1
3	**Ftpd_attacked**	[0.65 0.00 0.35]	[1.00 0.00 0.00]	176.9	-189.3
4	**Ftpd_attacked_detector**	[0.40 0.12 0.48]	[0.93 0.07 0.00]	165.8	-173.8
5	**Httpd_hacked**	[0.33 0.10 0.57]	[0.67 0.19 0.14]	197.4	-206.4
6	**Ftpd_hacked**	[0.12 0.00 0.88]	[0.96 0.00 0.04]	204.8	-203.5
7	**Website_defaced**	[0.33 0.33 0.33]	[0.33 0.33 0.33]	80.4	-80.0
8	**Webserver_sniffer**	[0.00 0.50 0.50]	[0.33 0.33 0.34]	716.3	-715.1
9	**Webserver_sniffer_detector**	[0.34 0.33 0.33]	[1.00 0.00 0.00]	148.2	-185.4
10	**Webserver_DOS_1**	[0.33 0.33 0.33]	[1.00 0.00 0.00]	106.7	-106.1
11	**Webserver_DOS_2**	[0.34 0.33 0.33]	[1.00 0.00 0.00]	96.5	-96.0
12	**Network_shut_down**	[0.33 0.33 0.33]	[0.33 0.33 0.33]	80.4	-80.0
13	**Fileserver_hacked**	[1.00 0.00 0.00]	[0.35 0.34 0.31]	1065.5	-1049.2
14	**Fileserver_data_stolen_1**	[1.00 0.00 0.00]	[1.00 0.00 0.00]	94.4	-74.0
15	**Workstation_hacked**	[1.00 0.00 0.00]	[0.31 0.32 0.37]	1065.5	-1049.2
16	**Workstation_data_stolen_1**	[1.00 0.00 0.00]	[1.00 0.00 0.00]	94.4	-74.0
17	**Fileserver_data_stolen_2**	[0.33 0.33 0.33]	[0.33 0.33 0.33]	80.4	-80.0
18	**Workstation_data_stolen_2**	[0.33 0.33 0.33]	[0.33 0.33 0.33]	80.4	-80.0

Table 1: Nash equilibrium strategies and state values for attacker and administrator

the two states that will lead him to the greatest damage to the network. When at these states, the attacker is just one state away from capturing the desired data from either the file server or the workstation. For the administrator, these two states have the most negative values (-1049.2), meaning most damage can be done to his network when it is in either of these states.

In state **Webserver_sniffer** (8^{th} row in Table 1), the attacker has a state value of 716.3, which is relatively high compared to those for other states. This is the state in which he has gained access to the public web server and installed a sniffer, i.e., a state that will potentially lead him to stealing the data that he wants. At this state, the value is -715.1 for the administrator. This is the second least desirable state for him.

6 Discussion

We could have modeled the interaction between the attacker and administrator as a purely competitive (zero-sum) stochastic game, in which case we would always find only a single unique Nash equilibrium. Modeling it as a general-sum stochastic game however, allows us to find potentially, multiple Nash equilibria. A Nash equilibrium gives the administrator an idea of the attacker's strategy and a plan for what to do in each state in the event of an attack. Finding more Nash equilibria thus allows him to know more about the attacker's best attack strategies. By using a stochastic game model, we are also able to capture the probabilistic nature of the state transitions of a network in real life. Solutions for stochastic models are however, hard to compute.

A disadvantage of our model is that the full state space can be extremely large. We are interested, however, in only a small subset of states that are in attack scenarios. One way of generating these states is the attack scenario generation method developed by Sheyner et al. [SJW02]. The set of scenario states can then be augmented with state transition probabilities and costs/rewards as functions of both players' actions so that our game-theoretic analysis can be applied. Another difficulty in our analysis is in building the game model. In reality, it may be difficult to quantify the costs/rewards for some actions and transition probabilities may not be easily available.

We note that the administrator's view of the game in our example is simplistic and uninteresting. This is because he only needs to act when he suspects the network is under attack. It is reasonable to assume the attacker and administrator both know what the other can each do. Such common knowledge affects their decisions on what action to take in each state and thus justifies a game formulation of the problem.

Finally, why not put in place all security measures? In practice, trade-offs have to be made between security and usability and a network may have to remain in operation despite known vulnerabilities (e.g., [Cru00]). Knowing that a network system is not perfectly secure, our game theoretic formulation of the security problem allows the administrator to discover the potential attack strategies of an attacker as well as best defense strategies against them.

7 Related Work

The use of game theory in modeling attackers and defenders has also appeared in several other areas of research. For example, in military and information warfare, the enemy is modeled as an attacker and has actions and strategies to disrupt the defense networks. Browne describes how static games can be used to analyze attacks involving complicated and heterogeneous military networks [Bro00]. In his example, a defense team has to defend a network of three hosts against an attacking team's *worms*. A defending team member can choose either to run a worm detector or not. Depending on the combined attack and defense actions, each outcome has different costs. This problem is similar to ours if we were to view the actions of each team member as separate actions of a single player. The interactions between the two teams, however, are dynamic, and can be better represented using a stochastic model like we did here. In his Master's thesis, Burke studies the use of repeated games with incomplete information to model attackers and defenders in information warfare [Bur99]. As in our work, the objective is to predict enemy strategies and find defenses against them using a game model. Using static game models, however, requires the problem to be abstracted to a very high level and only simple analyses are possible. Our use of a stochastic model in this paper allows us to capture the probabilistic nature of state transitions in real life.

In the study of network reliability, Bell considers a zero-sum game in which the router has to find a least-cost path and a network tester seeks to maximize this cost by failing a link [Bel01]. The problem is similar to ours in that two players are in some form of control over the network and that they have opposite objectives. Finding the least-cost path in their problem is analogous to finding a best defense strategy in ours. Hespanha and Bohacek discuss routing games in which an adversary tries to intersect data packets in a computer network [HB01]. The designer of the network has to find routing policies that avoid links that are under the attacker's surveillance. Finding their optimal routing policy is similar to finding the least-cost path in Bell's work [Bel01] and the best defense strategy in our problem in that at every state, each player has to make a decision on what action to take. Again, their game model a zero-sum game. In comparison, our work uses a more general (general-sum) game model which allows us to find more Nash equilibria.

McInerney et al. use a simple one-player game in their *FRIARS* cyber-defense decision system capable of reacting autonomously to automated system attacks [MSAH01]. Their problem is similar to ours in having cyberspace attackers and defenders. Instead of finding complete strategies, their single-player game model is used to predict the opponent's next move one at a time. Their model is closer to being just a Markov decision problem because it is a single-player game. Ours, in contrast, exploits fully what a game (two-player) model can allow us to find, namely, equilibrium strategies for both players.

Finally, Syverson talks about "good" nodes fighting "evil" nodes in a network and suggests using stochastic games for reasoning and analysis [Syv97]. In this paper, we have precisely formalized this idea and given a concrete example in detail. In summary, our work and example is different from previous work in that we employ a general-sum stochastic game model. This model allows us to perform a richer analysis for more complicated problems and also allows us to find multiple Nash equilibria (sets of best responses) instead of a single equilibrium.

8 Conclusions and Future Work

We have shown how the network security problem can be modeled as a general-sum stochastic game between the attacker and the administrator. Using the non-linear program NLP-1, we computed multiple Nash equilibria, each denoting best strategies (best responses) for both players. For one Nash equilibrium, we explained why these strategies make sense and are useful for the administrator. Discussions with one of our university's network managers revealed that these results are indeed useful. With proper modeling, the game-theoretic analysis we presented here can also be applied to other general heterogeneous networks.

In the future, we wish to develop a systematic method for decomposing large models into smaller manageable components such that strategies can be found individually for them using conventional Markov decision process (MDP) and game-theoretic solution methods such as dynamic programming, policy iteration, and value iteration. For example, nearly-isolated clusters of states can be regarded as subgames and states in which only one player has meaningful actions can be regarded as an MDP. The overall best-response for each player is then composed from the strategies for the components. We believe the computation time can be significantly reduced by using such a decomposition method. We also intend to use the method by Sheyner et al. [SJW02] for attack scenario generation to generate states so that we can experiment with network examples that are larger and more complicated. In our example, we manually enumerated the states for the attack scenario. The method in [SJW02] allows us to automatically generate the complete set of attack scenario states and thus allows us to perform a more complete analysis.

Acknowledgement

The first author is supported by the Singapore Institute of Manufacturing Technology (SIMTech) and the second

author, in part by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of SIMTech, the DOD, ARO, or the U.S. Government.

Bibliography

[Bel01] M.G.H. Bell. The measurement of reliability in stochastic transport networks. *Proceedings, 2001 IEEE Intelligent Transportation Systems*, pages 1183–1188, 2001.

[Bro00] R. Browne. C4I defensive infrastructure for survivability against multi-mode attacks. In *Proceedings, 21st Century Military Communications. Architectures and Technologies for Information Superiority*, volume 1, pages 417–424, 2000.

[Bur99] David Burke. Towards a game theory model of information warfare. Master's thesis, Graduate School of Engineering and Management, Airforce Institute of Technology, Air University, 1999.

[Cru00] Jeff Crume. *Inside Internet Security*. Addison Wesley, 2000.

[FV96] Jerzy Filar and Koos Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, New York, 1996.

[HB01] J.P Hespanha and S. Bohacek. Preliminary results in routing games. In *Proceedings, 2001 American Control Conference*, volume 3, pages 1904–1909, 2001.

[Mea01] C. Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1–2):143–164, 2001.

[MSAH01] J. McInerney, S. Stubberud, S. Anwar, and S. Hamilton. Friars: a feedback control system for information assurance using a markov decision process. In *Proceedings, IEEE 35th Annual 2001 International Carnahan Conference on Security Technology*, pages 223–228, 2001.

[SJW02] O. Sheyner, S. Jha, and J. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2002.

[Syv97] Paul F. Syverson. A different look at secure distributed computation. In *Proceedings, 10th Computer Security Foundations Workshop*, pages 109–115, 1997.

Modular Information Flow Analysis for Process Calculi

Sylvain Conchon

OGI School of Science & Engineering
Oregon Health & Science university
Beaverton, OR 97006 — USA
Sylvain.Conchon@cse.ogi.edu

Abstract

We present a framework to extend, in a modular way, the type systems of process calculi with information-flow annotations that ensure a noninterference property based on weak barbed bisimulation. Our method of adding security annotations readily supports modern typing features, such as polymorphism and type reconstruction, together with a *noninterference* proof. Furthermore, the new systems thus obtained can detect, for instance, information flow caused by contentions on distributed resources, which are not detected in a satisfactory way by using testing equivalences.

1 Introduction

Information flow analysis is used to guarantee secrecy and integrity properties of information. Traditionally this information is assigned a security level and the goal of the analysis is to prove that information at a given level never *interferes* with information at a lower level. Described in terms of *noninterference* [13], this turns out to be a dependency analysis [2] ensuring that *low level* outputs of a program do not depend on its *high level* information.

Much work in the security literature addresses this problem. Logical approaches have been used in [4, 5], while control flow analysis has been used in [6]. Recently, several studies have reformulated the problem as a typing problem, and a number of type systems have been developed to ensure secure information flow for imperative, sequential languages [28], functional ones [21, 14, 23, 24], imperative concurrent ones [19, 27, 7, 26], and process calculi [1, 16, 17, 22].

In type-based analysis, security classes are formalized as types and new type systems, combining information-flow-specific and standard type-theoretic aspects, are designed to guarantee the noninterference property. In the setting of process calculi, the *noninterference* result is expressed as a soundness result based on observational equivalence, which guarantees that a well-typed process $P[M]$, containing high level information M, *does not leak* M if $P[M]$ is indistinguishable from any well-typed process $P[M']$:

$$\text{if } \Gamma \vdash P[M] \text{ and } \Gamma \vdash P[M'] \text{ then } P[M] \approx P[M']$$

Different choices of equivalences may be used to state this noninterference result, and thus allow us to observe more or less the information flow from high level to low level. In this paper, we concentrate on bisimulation because testing equivalences are inadequate to observe information flow caused by contentions on distributed resources as shown in section 3.

Designing and proving information flow-aware type systems is non-trivial if we consider type systems providing complex features such as *polymorphism* and *type reconstruction*. To date, systems found in the type-based security literature are derived from existing one or built from scratch. Therefore, in order to reduce the proof effort involved in their design, only simple standard type-theoretic systems are usually defined, i.e monomorphic type systems, without type reconstruction or recursive types etc... (to our knowledge, only the type system of Pottier and Simonet [24] for the information flow analysis of core-ML is equipped with complex features.)

However, the drawback of these "monolithic" approaches [15] is that they suffer from modularity, since any modification of the standard part of the system requires the correctness of the whole system to be proved again. In this paper, we present a *modular* way to extend arbitrarily rich type systems for process calculi with security annotations and show their correctness, including noninterference, with a minimal proof effort. This framework allows us to design and prove a family of information flow-aware type system that provide complex features.

We formalize our work within the Join-Calculus [11], a named-passing process calculus related to the asynchronous π-calculus. We choose the Join-Calculus to formalize our framework because it enjoys a Hindley/Milner typing discipline with a family of rich polymorphic constraint-based type systems with type reconstruction [12, 8], which allows us to show how our method of adding security annotations readily supports modern typing features. However, because of its simplicity, we believe that our approach is applicable to other process calculi.

This work extends our previous work for the Lambda-Calculus [23]. However, in a distributed setting, the presence of concurrency and nondeterminism creates new information flows – disguised as control flows – which are more difficult to detect than those of a simple sequential programming language. We have to take into account, for instance, the non-termination of processes and the contentions on distributed resources as shown in the next section.

Our modular approach is based on a labelled mechanism which is used to track information dependencies throughout computations. This framework consists in two steps. First, we present an enriched Join-Calculus for which messages are labelled with a security level, and whose semantics propagates those security annotations **dynamically**. In that setting, the goal of the analysis is to prove that messages labelled with high-level security annotations never *interferes* with low-level ones. A *public* process is thus a process that sends only messages annotated with low-level security labels. We verify that the semantics propagates labels in a meaningful way by showing that it enjoys a noninterference property which guarantees that, if a *labelled* process $P[\mathbf{H} : M]$, containing high-level security information M –annotated with a label $\mathbf{H}$ – is *public* then, the *unlabelled* process $P[M]$ is *indistinguishable*, by using a weak barbed bisimulation (defined later), from any unlabelled process $P[M']$, provided that the labelled process $P[\mathbf{H} : M']$ is also public. The second step of our framework consists to approximate the propagation of labels **statically** by extending, in a *systematic* way, any type system of the Join-Calculus with security annotations. By defining a translation from the labelled Join-Calculus to the Join-Calculus, we show how to use existing type systems to analyze labelled processes, and how the soundness and noninterference proofs of the new system may rely upon, rather than duplicate, the correctness proofs of the original one.

The remainder of the paper is organized as follows. In the next section recall the syntax of the Join-Calculus and give its semantics using a standard reduction framework with evaluation contexts. In section 3, we compare the power of weak barbed bisimulation and may-testing equivalence to detect information flow on distributed systems. We present our approach in section 4 which is based on a labelled Join-Calculus whose syntax and semantics are formally defined in section 5. We show in section 6 that the semantics of this calculus yields a noninterference result based on bisimulation equivalence. Then, we show in section 7 how to translate our labelled calculus into the standard Join-Calculus, so as to use its type systems to analyze the flow of information of labelled processes (section 8). In section 9, we show how to define a more direct type system for the labelled Join-Calculus. We conclude in section 10.

2 The Join-Calculus

Let $\mathcal{N}$ be a countable set of *names* (also called channels) ranged over $u, v, x, y, \ldots$. We write $\vec{u}$ for a tuple $(u_1, \ldots, u_n)$ and $\bar{u}$ for a set $\{u_1, \ldots, u_n\}$, where $n \geq 0$. The syntax of the Join-Calculus is defined by the following grammar:

$$\begin{array}{lcl} P & ::= & 0 \mid (P \mid P) \mid u\langle \vec{v} \rangle \mid \texttt{def}\ D\ \texttt{in}\ P \\ D & ::= & J \triangleright P \mid D\ \texttt{or}\ D \\ J & ::= & u\langle \vec{x} \rangle \mid (J \mid J) \end{array}$$

A process P can be either the inert process 0, a parallel composition of processes $P \mid Q$, an asynchronous message $u\langle \vec{v} \rangle$ which sends a tuple of names $\vec{v}$ on a channel u, or a local channel definition which belongs to a process Q, written $\texttt{def}\ D\ \texttt{in}\ Q$. A definition D defines the receptive behavior of new channels. It is composed of several reaction rules $J \triangleright P$ that produce the process P whenever messages match the pattern J. In its simplest form, a join-pattern $u\langle \vec{x} \rangle$ waits for the reception of $\vec{x}$ on channel u. More generally join-patterns of the form $J \mid J$ express synchronization between co-defined channels.

We require all defined names in a join pattern to be pairwise different. The scoping rules of the calculus obey standard lexical bindings of functional languages. The only binder is the join-pattern which binds its formal parameters in the corresponding guarded process. New channels defined in $\texttt{def}\ D\ \texttt{in}\ P$ are bound in the main process P and recursively in every guarded process inside the definition D. The *defined names* $\mathrm{dn}(J)$ (resp. $\mathrm{dn}(D)$) of a join-pattern J (resp. of a definition D) are the channels defined by it. In a process $\texttt{def}\ D\ \texttt{in}\ P$, the defined names of D are bound within D and P. In the following, we assume that the set of free names and defined names of a process are always disjoint.

We define the operational semantics of the Join-Calculus using a standard reduction semantics with evaluation contexts (following [18] and [20]): *evaluation* contexts and *definition* contexts are defined by the following grammar:

$$\begin{array}{lcl} \mathcal{R} & ::= & [\,] \mid (P \mid \mathcal{R}) \mid (\mathcal{R} \mid P) \mid \texttt{def}\ D\ \texttt{in}\ \mathcal{R} \\ \mathcal{D} & ::= & [\,] \mid D\ \texttt{or}\ \mathcal{D} \mid \mathcal{D}\ \texttt{or}\ D \end{array}$$

We define a *structural* precongruence $\equiv$ over processes which satisfies the following axioms:

$$P \mid Q \equiv Q \mid P \qquad\qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

The reduction relation $\rightarrow$ is defined as the least precongruence which satisfies the following axioms:

$$(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \;\rightarrow\; \texttt{def}\ D\ \texttt{in}\ (P \mid Q)$$

$$\texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[J\sigma] \;\rightarrow\; \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[P\sigma]$$

where σ ranges over *renamings* from $\mathcal{N}$ into $\mathcal{N}$. The first rule needs the side condition $\mathrm{dn}(D) \cap \mathrm{fn}(Q) = \varnothing$ to avoid clashes between names. In the second rule, the names which appear in the join-pattern J are supposed to be not bound by the environment $\mathcal{R}$. These definitions differ from the standard definitions by restricting the structural relation to an associative and commutative relation, turning the scope extrusion law into an irreversible reduction rule and removing garbage collector rules which are not necessary to compute. Our operational semantics is also distinguished by the use of evaluation contexts to determine which messages can be "consumed" and replaced by the guarded process of a join-pattern isolated by a definition context. These modifications simplify the presentation of the semantics by removing the heating and cooling rules of definitions in the chemical style . In the following, we will write $\Rightarrow$ for the relation $(\equiv \cup \rightarrow)^\star$.

We define a basic notion of observation which detects the ability of a process to interact with its environment. In the Join-Calculus, the only way for a process to communicate is to emit a message on one of its free names. We thus define the *strong* predicate $\downarrow_u$ which detects whether a process emits on some free name u:

$$P \downarrow_u \stackrel{\text{def}}{=} \exists \mathcal{R}, \vec{v}.\ P = \mathcal{R}[u\,\langle\,\vec{v}\,\rangle] \quad \text{with } u \in \mathrm{fn}(P)$$

Then, we define the *may* predicate which detects whether a process may satisfy the basic observation predicate possibly after performing a sequence of internal reductions. We write $P \Downarrow_u$ if there exists a process P' such that $P \Rightarrow P'$ and $P' \downarrow_u$. From this predicate, we define the may-testing equivalence $\simeq$, which tests the may predicate under all possible evaluation contexts:

$$P \simeq Q \stackrel{\text{def}}{=} \forall \mathcal{R}[\,], u \in \mathcal{N}.\ \mathcal{R}[P] \Downarrow_u \ \text{ iff } \mathcal{R}[P] \Downarrow_u$$

This equivalence can also be refined by observing the internal choices of processes. We first define the weak barbed bisimulation (WB-bisimulation) $\stackrel{\bullet}{\approx}$ as the largest relation such that $P \stackrel{\bullet}{\approx} Q$ implies:

1. $P \Downarrow_u$ if and only if $Q \Downarrow_u$.
2. If $P \rightarrow P'$ then $\exists Q',\ Q \Rightarrow Q'$ and $P' \stackrel{\bullet}{\approx} Q'$
3. If $Q \rightarrow Q'$ then $\exists P',\ P \Rightarrow P'$ and $Q' \stackrel{\bullet}{\approx} P'$

Then, we define the *weak barbed congruence* $\approx$, which extends the previous relation with a congruence property.

$$P \approx Q \stackrel{\text{def}}{=} \forall \mathcal{R}[\,], u \in \mathcal{N}.\ \mathcal{R}[P] \stackrel{\bullet}{\approx} \mathcal{R}[Q]$$

3 WB-Bisimulation vs. May-Testing

We motivate the use of a WB-bisimulation to state our noninterference results with an example inspired by the class of attacks presented in [9] (see [10] for a classification of noninterference properties based on trace-based and bisimulation semantics).

This example describes an attack that compromises the privacy of a user's activities on the Web, by allowing any malicious Web site to determine the Web-browsing history of its visitor. This attack exploits the fact that Web browsers do not necessarily send HTTP requests to get pages that have been already visited by the user. We show through this example that these information leaks can only be observed in a satisfactory way by using a WB-bisimulation. Contrary to the *may*-testing, this equivalence allows us to observe the potential absence of HTTP requests.

Let us first review how Web caching works. Because the access of Web documents often takes a long time, Web browsers save copies of pages requested by the user, to reduce the response time of future accesses to those pages. However, browsers cannot save all the web pages requested by the user. The management of a cache consists in determining when a page should be cached and when it should be deleted. From a user point of view, it is difficult to anticipate the behavior of a browser when requesting a Web document. The browser may send an HTTP request to get a page even if it has been visited recently. We formalize the nondeterministic behavior of Web caching by the following Join-Calculus program:

$$\begin{array}{l} \texttt{def}\ get\,\langle\,url\,\rangle \ \triangleright\ http\,\langle\,url\,\rangle \\ \ \texttt{or}\ get\,\langle\,url\,\rangle \mid cache\,\langle\,url, page\,\rangle \triangleright\ page \end{array}$$

This program defines two communication channels get and $cache$. The user sends messages on channel get to get pages whose addresses are given as argument. Messages sent on channel $cache$ represent the cache entries which are composed of pairs of addresses and Web documents. The first definition $get\,\langle\,url\,\rangle \ \triangleright\ http\,\langle\,url\,\rangle$ waits for user messages and, upon receipt, sends HTTP requests to get the expected pages. The second definition $get\,\langle\,url\,\rangle \mid cache\,\langle\,url, page\,\rangle \triangleright\ page$ synchronizes messages sent on get and $cache$ whenever an entry in the cache matches a requested page, and returns the page to the user without requiring another Web access. These definitions are nondeterministic, so the browser may still send an HTTP request for a page that is saved in the cache.

We now present the attack we mentioned above. First of all, we assume that Internet communications are not secure. That is, any attacker is able to observe the HTTP requests sent by a browser.

Suppose that Alice visits Bob's Web site. Bob wants to find out whether his visitor has been to Charlie's Web site previously. For that, he writes an applet and embeds it in his home page. When Alice receives Bob's page, the applet is automatically downloaded and run on her browser. The applet tries to download Charlie's home page, and Bob observes whether the browser sends an HTTP request to get the page. If no request is sent, Bob concludes that Al-

ice has been to Charlie's site; if an HTTP message is sent, he learns nothing. We formalize Bob's attack by the program:

$$\begin{array}{l}\mathbf{Brw}[\underline{cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle}] \stackrel{\text{def}}{=}\\ \quad \mathtt{def}\ get\,\langle\,url\,\rangle\ \triangleright\ http\,\langle\,url\,\rangle\\ \quad\ \mathtt{or}\ get\,\langle\,url\,\rangle \mid cache\,\langle\,url, page\,\rangle \triangleright page\\ \quad \mathtt{in}\ \underline{cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle} \mid get\,\langle\,\mathtt{C}\,\rangle\end{array}$$

which describes the applet running on Alice's Web browser with Charlie's home page in the cache. We suppose that Alice has visited Charlie's site recently. The message sent on channel $cache$ corresponds to the cache entry which contains a copy of Charlie's home page (`C.html`). The message $get\,\langle\,\mathtt{C}\,\rangle$ encodes the request of the applet. The browser thus has the possibility to send a message on channel $http$ to get a fresh copy of the Charlie's home page or to return directly the page stored in the cache.

We can detect that the applet may obtain information stored in Alice's Web cache by the observation that the behavior of the program $\mathbf{Brw}[cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle]$ changes when the cache entry containing the home page of Charlie is replaced by, for instance, Denis's home page (`D.html`). In that case, the browser has *no choice* but to send an HTTP message to get the requested page.

The difference in behavior between these two programs can be made precisely by using an appropriate equivalence. First, we suppose that messages sent on channel $http$ are the only interactions observable by the environment, that is we only have the strong observation predicate $\downarrow_{\text{http}}$. Furthermore, we assume that the cache entries are the only secret information stored in the program.

When comparing these two programs with the may-testing equivalence, it turns out that they are indistinguishable since this equivalence does not allow us to observe the absence of HTTP requests: it only detects a successful interaction between a program and its environment. Thus, we have:

$$\mathbf{Brw}[cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle] \simeq \mathbf{Brw}[cache\,\langle\,\mathtt{D},\mathtt{D.html}\,\rangle]$$

On the contrary, when using the WB-bisimulation, we can observe the *internal* choice of the browser not to send an HTTP message. Thus, we obtain:

$$\mathbf{Brw}[cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle] \overset{\bullet}{\not\approx} \mathbf{Brw}[cache\,\langle\,\mathtt{D},\mathtt{D.html}\,\rangle]$$

This result shows that messages sent on channel $http$ may depend on high security level cache entries, and thus that the program $\mathbf{Brw}[cache\,\langle\,\mathtt{C}, C.html\,\rangle]$ is not a low security level program.

From a certain point of view, relying our noninterference on a bisimulation allows a particular kind of *timing leak* to be detected. However, this phenomenon is really limited, and our framework will does not detect such leak in general.

4 Our approach

In order to assert that a program P does not leak its secret information, we develop a dependency analysis that determines which messages of P contribute to its output. We prove the dependency analysis correct by showing that it yields a noninterference result based on WB-bisimulation.

Our approach is based on an analysis that tracks dependencies *dynamically* in the course of computation. For that, we extend the syntax of the Join-Calculus with labelled messages, and we equip its semantics with a mechanism to propagate labels. For instance, in our previous example, we would label the message sent on channel $cache$ with a high-level security annotation **H**, so as to determine its effects on the program's outputs.

$$\begin{array}{l}\mathtt{def}\ get\,\langle\,url\,\rangle\ \triangleright\ http\,\langle\,url\,\rangle\\ \ \mathtt{or}\ get\,\langle\,url\,\rangle \mid cache\,\langle\,url, page\,\rangle \triangleright page\\ \mathtt{in}\ \mathbf{H} : cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle \mid get\,\langle\,\mathtt{C}\,\rangle\end{array}$$

The basic idea to handle labels consists in propagating the labels of messages that match a definition D, to the process triggered by D. Unfortunately, this simple semantics would only lead to a noninterference result based on a *may*-testing equivalence which does not allow us to determine that the program $\mathbf{Brw}[\mathbf{H} : cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle]$ leaks its secret information. Indeed, this extended reduction rule does not allow us to propagate the label **H** of the cache entry to the message sent on channel $http$. Such a propagation would reveal the dependency of the $http$ channel upon $cache$.

Thus, in order to show the correctness of the dependency analysis with respect to a WB-bisimulation, we must also propagate the labels of messages that match a definition which is in contention with D, so as to track the dependencies due to the nondeterministic choices of a program.

Our solution to propagate these labels breaks the reduction rule of the Join-Calculus into two parts. The first part sets the labels of messages matching a definition up to the same label. In our example, the label **H** of the cache's entry can be propagated to the message on channel get so as to obtain a request $\mathbf{H} : get\,\langle\,\mathtt{C}\,\rangle$ which carries a high level security annotation. Thus, we have the following reduction where, by lack of space, we omit the contents of messages:

$$\begin{array}{l}\mathtt{def}\ get \mid cache \triangleright ..\\ \mathtt{in}\ \mathbf{H} : cache \mid get\end{array} \quad \mapsto \quad \begin{array}{l}\mathtt{def}\ get \mid cache \triangleright ..\\ \mathtt{in}\ \mathbf{H} : cache \mid \mathbf{H} : get\end{array}$$

The second rule performs the synchronization part of the reduction. It consumes messages carrying the same label, and propagates it to the process triggered by the definition. Therefore, in our example the browser can now synchronize the high-level request $\mathbf{H} : get\,\langle\,\mathtt{C}\,\rangle$ with the cache's entry $\mathbf{H} : cache\,\langle\,\mathtt{C},\mathtt{C.html}\,\rangle$ and return the page **H**:`C.html`. Furthermore, it can also send an HTTP request $\mathbf{H} : http\,\langle\,\mathtt{C}\,\rangle$ annotated with a label **H** showing its dependencies on the secret cache entry. Thus, we have:

$$\begin{array}{lcl} \texttt{def}\ get \triangleright .. & & \texttt{def}\ ..\ \texttt{in}\ \mathbf{H} : \texttt{C.html} \\ \texttt{or}\ get \mid cache \triangleright .. & \nearrow & \\ \texttt{in}\ \mathbf{H} : cache \mid \mathbf{H} : get & \rightarrow & \texttt{def}\ ..\ \texttt{in}\ \mathbf{H} : http \end{array}$$

Unfortunately, these rules do not track dependencies along all the executions of a program. In our example, the browser has the possibility of answering the applet's request $get\,\langle\,\texttt{C}\,\rangle$ by sending directly an unlabelled HTTP message $http\,\langle\,\texttt{C}\,\rangle$. That is,

$$\begin{array}{lcl} \texttt{def}\ get \triangleright .. & & \texttt{def}\ get \triangleright .. \\ \texttt{in}\ \mathbf{H} : cache \mid get & \rightarrow & \texttt{in}\ \mathbf{H} : cache \mid http \end{array}$$

Nevertheless, it is sufficient to assert, for instance, that the program $\mathbf{Brw}[cache\,\langle\,\texttt{C}, \texttt{C.html}\,\rangle]$ runs at a **high** security clearance. Furthermore, we can prove that this extended semantics enjoys the noninterference result based on WB-bisimulation.

5 A labelled Join-Calculus

In this section we define formally our Join-Calculus with labelled messages. Labels, ranged over by small Greek letters $\alpha, \beta, \ldots$, are supposed to be taken from an upper semi-lattice $(\mathcal{L}, \leq, \sqcup, \bot)$, and are used to indicate the *security* level of messages. The syntax of the labelled Join-Calculus is defined as follows:

$$P ::= \ldots \mid \alpha : u\,\langle\,\vec{v}\,\rangle \mid \ldots$$

(in the following, we will write $\prod_{i\in I}(P_i)$ for the parallel composition of the processes P_i)

For simplicity, the syntax of our labelled calculus does not include labelled processes but only messages. Intuitively, a labelled process $\alpha : P$ is a process which interacts with its environment at a security clearance α. That is, all barbs of the process have a security annotation of at least α. To recover the construction $\alpha : P$, we arrange that every message of P is annotated by a label greater than or equal to α. Formally, we define a function $\alpha \bullet P$ that performs the propagation of a label α on a process P:

$$\begin{array}{c} \alpha \bullet 0 = 0 \qquad \alpha \bullet (P \mid Q) = (\alpha \bullet P) \mid (\alpha \bullet Q) \\ \alpha \bullet (\beta : u\,\langle\,\vec{v}\,\rangle) = (\alpha \sqcup \beta) : u\,\langle\,\vec{v}\,\rangle \qquad \alpha \bullet u\,\langle\,\vec{v}\,\rangle = \alpha : u\,\langle\,\vec{v}\,\rangle \\ \alpha \bullet (\texttt{def}\ D\ \texttt{in}\ P) = \texttt{def}\ D\ \texttt{in}\ \alpha \bullet P \end{array}$$

This definition restricts the propagation of α to messages which are not guarded by join-patterns. In the following, we will say that a process P *runs* at a security clearance α, if all its active messages, that is those not guarded by join-patterns, are annotated with a label greater than or equal to α. The evaluation and definition contexts of the labelled calculus are those defined for the Join-Calculus, as well as the structural relation $\equiv$. Its operational semantics is now composed of two reductions $\rightarrow$ and $\mapsto$ to manage security annotations. The relation $\rightarrow$ is defined as the least precongruence which satisfy the following two axioms:

$$\begin{array}{c} (\texttt{def}\ D\ \texttt{in}\ P) \mid Q \rightarrow \texttt{def}\ D\ \texttt{in}\ (P \mid Q) \\ \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\alpha \bullet J\sigma] \rightarrow \\ \texttt{def}\ \mathcal{D}[J \triangleright P]\ \texttt{in}\ \mathcal{R}[\alpha \bullet P\sigma] \end{array}$$

which expect the same side conditions than those of section 2. The relation $\mapsto$ is defined by the following axiom:

$$\begin{array}{lcl} \texttt{def}\ \mathcal{D}[J \triangleright P] & & \texttt{def}\ \mathcal{D}[J \triangleright P] \\ \texttt{in}\ \mathcal{R}[\prod_{i\in I}(\alpha_i : u_i\,\langle\,\vec{v}_i\,\rangle)] & \mapsto & \texttt{in}\ \mathcal{R}[\prod_{i\in I}(\alpha : u_i\,\langle\,\vec{v}_i\,\rangle)] \end{array}$$

where the join-pattern J has the form $\prod_{i\in I}(u_i\,\langle\,\vec{x}_i\,\rangle)$ and the names u_i are not bound by $\mathcal{R}$. Furthermore, the label α is equal to $\bigsqcup_{i\in I}\alpha_i$ and there exists a substitution σ such that $\forall i \in I,\ \sigma(\vec{x}_i) = \vec{v}_i$.

The basic reduction step $\rightarrow$ extends the reduction of the Join-Calculus so as to propagate labels throughout the computation. The guarded process P of a join-pattern J is substituted for a parallel composition of messages, $\alpha \bullet J\sigma$, that matches J and for which each message carries the *same* label α. The actual names are then substituted for the formal parameters, and the label α is *propagated* on P, so as to guarantee that any observation of P will carry a label, at least, greater than or equal to α. This records the fact that all these barbs depend on the messages that have been used to trigger the process P. This propagation mechanism explains the restriction, in the function $\alpha \bullet P$, to broadcast the label α only to messages of P which are not guarded by join-patterns. Since a reduction step propagates the labels of messages consumed by a join-pattern to its guarded process, every message produced by a reduction will necessarily be annotated with a label greater than or equal to α. Furthermore, following the semantics of the Join-Calculus, scope extrusion is turned into a reduction rule, thus making it irreversible.

The second reduction rule $\mapsto$ sets the labels of messages which match a join-pattern, up to the same level. According to the basic reduction step, the reduction of a parallel composition of messages $\prod_{i\in I}(\alpha_i : u_i\,\langle\,\vec{v_i}\,\rangle)$ requires the labels α_i to be equal. The reduction $\mapsto$ is thus used to promote the labels α_i to the least upper bound $(\sqcup_{i\in I}\alpha_i)$. In the following, we will use the notations $\Rightarrow$ for $(\equiv \cup \rightarrow)^\star$ and $\rightsquigarrow$ for $(\equiv \cup \rightarrow \cup \mapsto)^\star$.

For technical reasons, we define an auxiliary *structural* precongruence on processes, noted $\asymp$, which equips the labelled Join-Calculus with garbage collection rules that allow the removal of inert processes and local definitions whose channels are not referenced. In the following, we will say that a process P is *gc-equivalent* to a process Q if $P \asymp Q$. This relation will be used to state our noninterference result. It satisfies the following axioms:

$$\begin{array}{c} P \mid 0 \asymp P \qquad 0 \mid P \asymp P \\ \texttt{def}\ D\ \texttt{in}\ P \asymp P \quad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(P) = \emptyset \end{array}$$

Finally, we extend the observation predicates of the Join-Calculus to deal with labels. The basic observation pred-

icate is improved so as to detect the security level of the strong barbs of a process:

$$P \downarrow_{\alpha:u} \stackrel{\text{def}}{=} P = \exists \mathcal{R}, \vec{v}.\, \mathcal{R}[\alpha : u\,\langle \vec{v} \rangle] \quad \text{with } u \in \text{fn}(P)$$

We extend the weak barb predicate accordingly.

6 Noninterference up to WB-Bisimulation

We prove in this section that the semantics of the labelled Join-Calculus yields a noninterference result based on WB-bisimulation. For simplicity, we assume a fixed security lattice $\mathcal{L}$ consisting of the set $\{\mathbf{L}, \mathbf{H}\}$, for "low" and "high" security information, and ordered by $\mathbf{L} \leq \mathbf{H}$. However, all the results in this paper would carry through for a richer lattice.

Definition 1 *A process P is a* public *process if all its weak barbs are annotated with the label* $\mathbf{L}$.

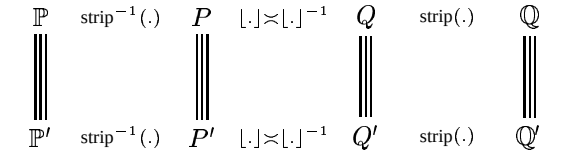
Let $\lfloor . \rfloor$ be a homomorphism which takes as argument a process P, and returns a process where every message carrying a label $\mathbf{H}$ has been replaced by the inert process 0, and $\text{strip}(P)$ be the process obtained by removing every label in the process P. The following result shows that our labelled Join-Calculus enjoys a noninterference property based on WB-bisimulation, which guarantees that a *public* process doesn't leak its *high* security level information.

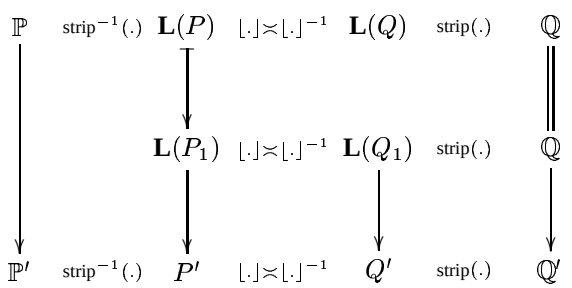
Theorem 1 (Dynamic Noninterference) *If P and Q are two* public *processes such that $\lfloor P \rfloor \asymp \lfloor Q \rfloor$, then strip$(P) \stackrel{\bullet}{\approx}$ strip(Q).*

Proof. The core of the proof consists to prove that the relation

$$\{(\mathbb{P}, \mathbb{Q})\,;\, \lfloor P \rfloor \asymp \lfloor Q \rfloor \wedge \mathbf{L}(P) \wedge \mathbf{L}(Q)\}$$

with $\mathbb{P} = \text{strip}(P)$ and $\mathbb{Q} = \text{strip}(Q)$, is a barbed bisimulation (we use the notation $\mathbf{L}(P)$ to state that a process P is *public*). For that, we show that the following diagrams commute.





Then, $\mathbb{P} \Downarrow_u$ implies that there exists a process $\mathbb{P}'$ such that $\mathbb{P} \Rightarrow \mathbb{P}'$ and $\mathbb{P}' \downarrow_u$. Following the previous diagrams, we deduce that there exists a process P' and a label α such that $\mathbb{P}' = \text{strip}(P')$ and $P' \downarrow_{\alpha:u}$. Furthermore, since P' is a public process, we have $\alpha = \mathbf{L}$ and there exists a process $\mathbb{Q}'$ such that $\mathbb{Q} \Rightarrow \mathbb{Q}'$ and $\mathbb{Q}' = \text{strip}(Q')$ and $\lfloor P' \rfloor \asymp \lfloor Q' \rfloor$ and $\mathbf{low}(Q')$. So, since $P' \downarrow_{\mathbf{L}:u}$, there exists an environment $\mathcal{R}[\ ]$ such that $P' = \mathcal{R}[\mathbf{L} : u\,\langle \vec{v} \rangle]$ and we can show that $\lfloor P' \rfloor \asymp \lfloor Q' \rfloor$ implies that there exists a environment $\mathcal{R}'[\ ]$ such that $Q' = \mathcal{R}'[\mathbf{L} : u\,\langle \vec{v} \rangle]$ and $u \in \text{fn}(Q')$. Therefore, we have $Q' \downarrow_{\mathbf{L}:u}$ that is $\mathbb{Q}' \downarrow_u$ and $\mathbb{Q} \Downarrow_u$. The result follows be the symmetry of the relation.

This theorem guarantees that if two public labelled processes P and Q differ only on high level information (i.e. if $\lfloor P \rfloor \asymp \lfloor Q \rfloor$) then, the unlabelled processes $\text{strip}(P)$ and $\text{strip}(Q)$ cannot be distinguished by using a WB-bisimulation. It is important to note that the well-foundedness of our propagation rules is implied by the fact that the equivalence $\text{strip}(P) \stackrel{\bullet}{\approx} \text{strip}(Q)$ only relies on standard Join-Calculus processes.

Now, in order to ensure secure information flow *statically* rather than *dynamically*, we define a decidable type system for the labelled Join-Calculus that approximates its semantics. However, instead of building our system from scratch – or by modifying an existing type system – we define an encoding from our labelled calculus to the unlabelled one, so as to use standard type systems of the Join-Calculus to analyze the processes produced by the translation. Then, by composing this translation with an existing type system – which may provide for instance polymorphism and type reconstruction – we can define a variety of information flow aware type systems whose soundness and noninterference properties rely only on the soundness proof of the original system, and are thus proved almost for free.

7 Translation

Our target calculus is the Join-Calculus with a set of names supplemented with *label* constants that represent *security labels*, taken from the security lattice $\mathcal{L}$, and a primitive, $@$, used to concatenate labels:

$$u, v ::= n \mid \alpha \mid u@v \qquad (\alpha \in \mathcal{L})$$

In the following, we write $\bigoplus_{i \in I} u_i$ for the concatenation of the names u_i. We extend the operational semantics of the Join-Calculus with a delta rule which states that $@$ returns the least upper bound of its arguments:

$$\alpha@\beta \rightarrow \alpha \sqcup \beta \qquad \forall \alpha, \beta \in \mathcal{L}$$

The translation function from the labelled Join-Calculus to the Join-Calculus is defined by the rules in figure 1.

$$[\![0]\!] = 0 \qquad [\![\alpha : u\,\langle \vec{v} \rangle]\!] = u\,\langle \alpha, \vec{v} \rangle$$
$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \qquad [\![D_1 \texttt{ or } D_2]\!] = [\![D_1]\!] \texttt{ or } [\![D_2]\!]$$
$$[\![\texttt{def } D \texttt{ in } P]\!] = \texttt{def } [\![D]\!] \texttt{ in } [\![P]\!]$$

$$\left[\!\!\left[\prod_{i\in I} (u_i\,\langle \vec{x}_i \rangle) \triangleright P \right]\!\!\right] = \begin{cases} \prod_{i\in I} (u_i\,\langle \mathtt{pc}_i, \vec{x}_i \rangle) \triangleright \prod_{i\in I} \left(u_i \left\langle \mathop{@}_{i\in I} \mathtt{pc}_i \;,\; \vec{x}_i \right\rangle \right) \\ \texttt{or } \prod_{i\in I} (u_i\,\langle \mathtt{pc}, \vec{x}_i \rangle) \triangleright \; \mathtt{pc} \star [\![P]\!] \end{cases}$$

for fresh names $\mathtt{pc}, \mathtt{pc}_1, \ldots, \mathtt{pc}_n$.

Figure 1: Translation from the labelled Join-Calculus to the Join-Calculus

The basic idea of the encoding is to use the polyadicity of the target calculus to map every labelled message $\alpha : u\,\langle \vec{v} \rangle$ to the unlabelled one $u\,\langle \alpha, \vec{v} \rangle$, whose first component is the security level of the labelled message. Following this basic idea, every join-pattern $\prod_{i\in I} (u_i\,\langle \vec{x}_i \rangle) \triangleright P$ is mapped to a pair of join-patterns which define the same channels with an extra argument representing the security level of messages sent on u_i. The first join-pattern is used to promote the security levels $\mathtt{pc}_i$ of each message sent on channel u_i to the concatenation of all security levels. The second join-pattern propagates explicitly the security level argument $\mathtt{pc}$ into $[\![P]\!]$, so that every message of $[\![P]\!]$ has its first argument concatenated with $\mathtt{pc}$. The messages sent on channels u_i must match the same security level $\mathtt{pc}$ since, according to the basic reduction step of the labelled Join-Calculus, the messages consumed by a join-pattern must all carry the same label. The propagation function used in the translation is similar to the function defined in section 5: it propagates $\mathtt{pc}$ to messages which are not guarded by join-patterns and which carry at least one argument:

$$\mathtt{pc} \star 0 = 0 \qquad \mathtt{pc} \star u\,\langle \alpha, \vec{v} \rangle = u\,\langle \mathtt{pc}@\alpha, \vec{v} \rangle$$
$$\mathtt{pc} \star (P \mid Q) = (\mathtt{pc} \star P) \mid (\mathtt{pc} \star Q)$$
$$\mathtt{pc} \star (\texttt{def } D \texttt{ in } P) = \texttt{def } D \texttt{ in } \mathtt{pc} \star P$$

The rest of the rules define the translation function as a homomorphism on processes. We prove the correctness of our encoding by showing that the process $[\![P]\!]$ mimics the reduction steps of the process P.

Lemma 1 (Simulation) *If $P \rightsquigarrow P'$ then $[\![P]\!] \Rightarrow [\![P']\!]$.*

8 Typing the Labelled Join-Calculus

By composing our encoding with a standard type system of the Join-Calculus, we can define a type system for our labelled calculus such that a "labelled" process is well-typed if and only if its translation is well-typed. That is,

$$\Gamma \vdash P \text{ holds if and only if } \Gamma \vdash [\![P]\!]$$

where typing judgments of the form $\Gamma \vdash P$ states that a process P is well-typed under the assumption Γ, which is a set of bindings of the form $u : t$ (where t belongs to a set of type T). In practice, we can pick any type system of the Join-Calculus that guaranteeing the following requirements:

1. (Compositionality) If $\Gamma \vdash \mathcal{R}[P]$ then $\Gamma \vdash P$.
2. (Subject Reduction) If $\Gamma \vdash P$ and $P \Rightarrow Q$ then $\Gamma \vdash Q$.
3. (Labels) Every label is a type: $\mathcal{L} \subseteq T$. If $\alpha, \beta \in \mathcal{L}$ and $\Gamma \vdash \alpha : \beta$ then $\alpha \leq \beta$.
4. (Messages) If the assumption on the type of a name u is $\Gamma(u) = \langle \vec{t} \rangle$ (which means that u is a channel expecting a tuple of arguments of type $\vec{t}$) then, the typing judgment $\Gamma \vdash u\,\langle \vec{v} \rangle$ holds if and only if $\Gamma \vdash \vec{v} : \vec{t}$ holds.

Then, it is easy to show that the new type system thus obtained enjoys the following soundness result, whose proof only relies on the simulation property and the axioms given above.

Theorem 2 *If $\Gamma \vdash P$ and $P \rightsquigarrow Q$ then $\Gamma \vdash Q$.*

Proof. By definition, $\Gamma \vdash P$ may be read $\Gamma \vdash [\![P]\!]$. Furthermore, according to the simulation property, $P \rightsquigarrow Q$ implies $[\![P]\!] \Rightarrow [\![Q]\!]$. Thus, because we suppose that the source type system enjoys a subject reduction property (axiom 2), we have $\Gamma \vdash [\![Q]\!]$ that is $\Gamma \vdash Q$.

We now show that the new type system enjoys a noninterference property, called *static* noninterference, whose proof relies on the *dynamic* noninterference theorem and the soundness proof above. A environment Γ is *public* when $\Gamma = (u_i : \langle \mathbf{L}, \vec{t_i} \rangle)^{i\in I}$.

Theorem 3 (Static Noninterference) *If P and Q are two processes such that $\lfloor P \rfloor \asymp \lfloor Q \rfloor$ then, if there exists a* public *environment Γ such that $\Gamma \vdash P$ and $\Gamma \vdash Q$ hold then, we have strip$(P) \mathrel{\dot{\approx}}$ strip(Q).*

Proof. We just have to show that P and Q are both *public*. Assume that $P \Downarrow_{\alpha:u}$ (the case for Q is similar) then, there exists a context $\mathcal{R}$ such that $P \rightsquigarrow \mathcal{R}[\alpha : u\,\langle \vec{v} \rangle]$ and u is not bound in $\mathcal{R}$. According to theorem 2, $\Gamma \vdash \mathcal{R}[\alpha : u\,\langle \vec{v} \rangle]$ holds, which may be read $\Gamma \vdash [\![\mathcal{R}[\alpha : u\,\langle \vec{v} \rangle]]\!]$, that is $\Gamma \vdash [\![\mathcal{R}]\!][u\,\langle \alpha, \vec{v} \rangle]$. Then, by compositionality (axiom 1), we have $\Gamma \vdash u\,\langle \alpha, \vec{v} \rangle$. According to our hypothesis on Γ and to axioms 3 and 4, it follows that $\alpha = \mathbf{L}$. So, P is a *public* process and the result follows by theorem 1.

This noninterference property states that typing judgments of the new system provide a correct approximation of the semantics of the labelled Join-Calculus. Again, we assume a simple dichotomy between *low* and *high* security levels, reducing the security lattice $\mathcal{L}$ to the set $\{\mathbf{L}, \mathbf{H}\}$. Recovering a noninterference result for an arbitrary security lattice can be done very easily. Furthermore, the interesting aspect of our noninterference proof is that it only relies on the noninterference property proved in section 6 and on the soundness proof of the original system. The *modularity* of the proof allows us to modify the standard part of the type system without having to prove its noninterference property again. Actually, this is the main advantage of our propagation rules: we prove a *dynamic* noninterference result once and we obtain a set of *static* noninterference results for a variety of type systems.

As a corollary, we can easily prove a noninterference result based on the weak barbed congruence, provided we admit only well-typed contexts. This congruence, noted $\overset{\vdash}{\approx}$ is defined such that $P \overset{\vdash}{\approx} Q$ implies that for all context $\mathcal{R}[\,]$ and typing environment Γ, if $\Gamma \vdash \mathcal{R}[P]$ and $\Gamma \vdash \mathcal{R}[Q]$ then $\text{strip}(\mathcal{R}[P]) \overset{\bullet}{\approx} \text{strip}(\mathcal{R}[Q])$.

Corollary 1 *Let P and Q be two processes such that $\lfloor P \rfloor \asymp \lfloor Q \rfloor$, then if there exists a public environment Γ such that $\Gamma \vdash P$ and $\Gamma \vdash Q$ hold then, $P \overset{\vdash}{\approx} Q$.*

Proof. Let $\mathcal{R}[\,]$ a evaluation context and Δ a typing environment such that $\Delta \vdash \mathcal{R}[\text{strip}(P)], \Delta \vdash \mathcal{R}[\text{strip}(Q)]$. From $\mathcal{R}[\,]$, we may obtain a labelled context $\mathcal{R}_{\mathbf{L}}[\,]$ with every message annotated with the labelled $\mathbf{L}$. We can then easily show that the environment Δ can be extended in a public environment $\Delta_{\mathbf{L}}$ (construct from Δ and Γ) such that $\Delta_{\mathbf{L}} \vdash \mathcal{R}_{\mathbf{L}}[P], \mathcal{R}_{\mathbf{L}}[Q]$.Then, since $\text{strip}(\mathcal{R}_{\mathbf{L}}[\,]) = \mathcal{R}[\,]$, the result follows by applying the theorem 3.

9 A concrete example

The new type systems defined by this approach are as *expressive* as the original ones. We may obtain for instance, almost for free, a noninterference proof for a *family* of rich constraint-based type system – with type reconstruction – by using the generic framework JOIN(X) presented in [8].

We illustrate our approach with a basic polymorphic type system with subtyping called B(T) defined in [8]. The results shown, however, do not rely on any specific properties of B(T), i.e. any type systems that satisfies the property defined in the previous section could be used.

B(T) is a *ground* type system: it does not have a notion of type variable. Instead, it has *monotypes*, denoted by t, taken to be elements of some set T, and *polytypes*, denoted by s, merely defined as certain subsets of T.

The set T is a parameter of the type system, and can be instantiated by any set equipped with a partial order $\preccurlyeq$ and a total function, denoted $\langle\cdot\rangle$, from $T^\star$ into T, such that $\langle \vec{t} \rangle \preccurlyeq \langle \vec{t'} \rangle$ holds if and only if $\vec{t} \preccurlyeq \vec{t}$. Furthermore, we will let S denote the set of all polytypes.

Monotype environments, denoted by $\mathcal{B}$, are sets of bindings of the form $u : t$, while *polytype* environments, denoted by Γ or $\mathcal{A}$, associates names with polytypes. Given two monotype environments $\mathcal{B}$ and $\mathcal{B}'$, we define $\mathcal{B} + \mathcal{B}'$ as the monotype environment which maps every $u \in \mathcal{N}$ to $\mathcal{B}'(u)$, if it is defined, and to $\mathcal{B}(u)$ otherwise. When $\mathcal{B}(u) = \mathcal{B}'(u)$ for all names u that $\mathcal{B}$ and $\mathcal{B}'$ both define then, $\mathcal{B} + \mathcal{B}'$ is written $\mathcal{B} \oplus \mathcal{B}'$. Finally, we define the Cartesian product $\Pi\mathcal{A}$ of a polytype environment $\mathcal{A} = (u_i : s_i)^{i \in I}$, as the set of tuples $\{(u_i : t_i)^{i \in I} \mid \forall i \in I,\ t_i \in s_i\}$.

We now define a type system for the labelled Join-Calculus as an instance of B(T) with a set T of monotypes inductively generated by:

$$\mathcal{L} \subseteq T \quad \text{and} \quad \vec{t} \in T \text{ implies } \langle \vec{t} \rangle \in T$$

and partially ordered by $\preccurlyeq$:

$$\alpha \leq \beta \text{ implies } \alpha \preccurlyeq \beta \quad \text{and} \quad \vec{t} \preccurlyeq \vec{t'} \text{ implies } \langle \vec{t'} \rangle \preccurlyeq \langle \vec{t} \rangle$$

This instance, called B($\mathcal{L}$), is defined by the rules defined in figure 2. We distinguish three kinds of typing judgments:

- $\Gamma \vdash u : t$ states that the name u has type t under assumptions Γ.
- $\Gamma \vdash D :: \mathcal{B}$ (resp. $D :: \mathcal{A}$) states that the definition D gives rise to the environment fragment $\mathcal{B}$ (resp. $\mathcal{A}$) under assumptions Γ.
- $\Gamma \vdash P$ states that the process P is well-typed under assumptions Γ.

Furthermore, every typing environment Γ of B($\mathcal{L}$)'s judgments contains an initial set of bindings $(\alpha : \alpha)^{\alpha \in \mathcal{L}}$ which state that every label constant is also a type.

The most interesting aspect of the system B($\mathcal{L}$) is its *extensional* view of polymorphism: a polytype s is by definition equivalent to the set of its monotype instances. The rule INST performs *instantiation* by allowing a polytype s to be specialized to any monotype $t \in s$, while the rule GEN performs *generalization* by allowing the judgment $\Gamma \vdash D :: (u_i : s_i)^{i \in I}$ to be formed if the judgments $\Gamma \vdash D :: (u_i : t_i)^{i \in I}$ hold for all $(u_i : t_i)^{i \in I} \in \Pi(u_i : s_i)^{i \in I}$.

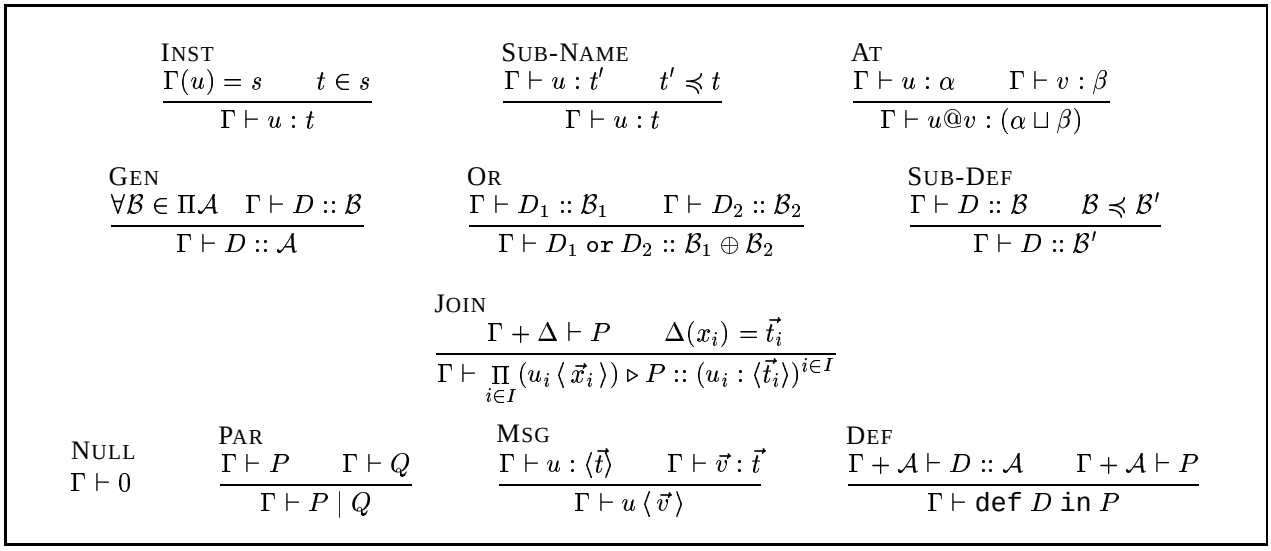


Figure 2: The system $\mathrm{B}(\mathcal{L})$

Typing rules for processes, except rule DEF, are similar to those found in common typed process calculi. The rule DEF and those for typing definitions, are inspired by the typing rules of ML extended with *polymorphic recursion* and *subtyping*. The system is showed in [8] to enjoy the following soundness result.

Theorem 4 (Subject Red.) *If* $\Gamma \vdash P$ *and* $P \Rightarrow P'$ *then* $\Gamma \vdash P'$.

Since $\mathrm{B}(\mathcal{L})$ respects all the axioms listed in section 8, it could be used to define an "indirect" type system for the labelled Join-Calculus that enjoys the properties defined in section 8.

However, we can give a more direct description of this type system by systematically composing the translation function defined in section 7 with the $\mathrm{B}(\mathcal{L})$'s rules. We obtain the rules of figure 3 which are very similar to those of $\mathrm{B}(\mathcal{L})$. The main difference rests on the typing judgments for processes which now have the form $\Gamma \vdash_{\mathtt{pc}} P$, where pc is a security level used to enforce the security level of P. Intuitively, such a judgment may be read:

> "*P is well typed under Γ and it will only emit messages annotated with a security label of at least* pc"

Furthermore, channel types of the form $\langle \mathtt{pc}, \vec{t} \rangle$ have been replaced with $\langle \vec{t} \rangle^{\mathtt{pc}}$, so as to insist on the fact that we are dealing with types carrying security annotations.

The main use of the security level pc is in the rules D-MSG, D-JOIN and D-SUB-PC. The D-MSG enforces the security clearance of a process. It requires the channels' security level to match the level pc attained by the process. Furthermore, it restricts the security level of its messages to be at most pc, so as to prevent direct flows of information. The D-JOIN requires the channel names of a definition to have the same security level. This reflects the reduction rule of the labelled Join-Calculus which allows only to reduce messages carrying the same label. The process P triggered by the definition is thus typed at a security level pc so as to reflect the propagation of labels carrying by messages which match a definition. Finally, the D-SUB-PC is used to increase the security level of a process.

These rules seem quite intuitive and it would have been easy to come up directly without using our framework. However, the main advantage of our approach rests on the *systematic* way of deducing a security aware type system from a standard one. Contrary to other approaches, our typing rules described formal semantic properties of programs, leaving no doubt that our design decisions are natural.

10 Conclusion and related work

As illustrated by Sabelfeld and Sands [25, 26], the definition of a semantic-based model of information-flow provides a fair degree of modularity which facilitates the correctness proofs of security type systems. We have shown in this paper how to extend, in a *modular* way and with a *minimal proof effort*, standard type systems of the Join-Calculus to ensure secure information flow. The main advantage of our approach rests on a labelled mechanism that track dependencies throughout computations. By translating the semantics of this mechanism to the semantics of the Join-Calculus, we may extend standard type systems to ensure information flow of labelled processes. Furthermore, the new systems obtained are as expressive as the original ones. We may prove, for instance, a noninterference proof – relying on WB-bisimulation – for rich polymorphic type systems almost for free.

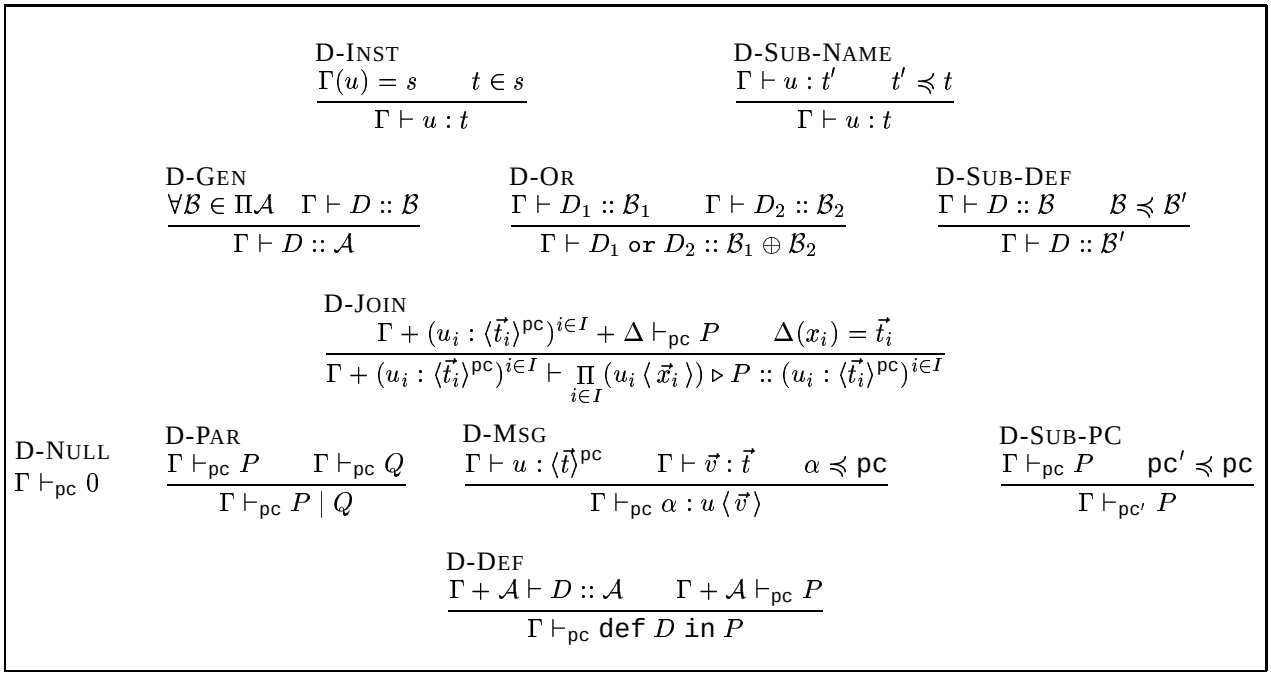


Figure 3: A direct type system derived from B($\mathcal{L}$)

The question of information flow analysis in the setting of process calculi has been studied previously in [10, 1, 16, 17, 22]. The last four papers investigate the use of type systems to ensure the noninterference property. Hennessy and Riely [16] study an *asynchronous* π-calculus extended with security annotations on processes and channels, and prove a noninterference property based on *may*-testing equivalence. The may-testing semantics is, in our eyes, too coarse since it does not allow one to detect, for instance, the information flow of the browser program described in section 3. Pottier shows in [22] how to reduce the problem of proving noninterference in the π-calculus to the subject-reduction proof of a type system for an extension of this calculus, named the $\langle\pi\rangle$-calculus. This calculus describes the independent execution of a pair of processes (which shared some sub-processes), and allows to prove a noninterference based on WB-bisimulation. Honda *et al.* propose in [17] an advanced type system with linearity and affinity informations used to relax the restrictions on information flow when linear channels are involved in communications. The perspective of extending our framework with linearity information is an interesting issue. Abadi studies in [1], a monomorphic type system which ensures secrecy properties for cryptographic protocols described in a variant of the π-calculus, called the spi-calculus. The main interest of this system relies on the typing rules for encryption/decryption primitives of the spi-calculus, that treats encryption as a form of safe declassification. Following this work, it would be interesting to extend our approach to the SJoin-Calculus [3] an extension of the Join-Calculus with constructs for public-key encryption proposed by the same author.

Acknowledgements

We would like to thank François Pottier, Alan Schmitt and Daniel de Rauglaudre for stimulating and insightful discussions, James Leifer and Jean-Jacques Lévy for their help to make the paper clearer, and the anonymous referees for numerous suggestions for improvements.

Bibliography

[1] M. Abadi. Secrecy by typing in security protocols. In *14th Symposium on Theoretical Aspects of Computer Science (STACS'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[2] M. Abadi, A. Banerjee, N. Heintze, and J.G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. IEEE, Computer Society Press, July 1998.

[4] G.R. Andrews and R.P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans-*

actions on Programming Languages and Systems, 2(1):56–76, January 1980.

[5] J-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings of the 3rd European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–74, 1994.

[6] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis of processes for no read-up and no write-down. *Lecture Notes in Computer Science*, 1578:120–134, 1999.

[7] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *The 28th International Colloquium on Automata, Languages and Programming (ICALP) Crete, Greece, July 8-12, 2001*, July 2001.

[8] S. Conchon and F. Pottier. JOIN(X): Constraint-based type inference for the join-calculus. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 221–236. Springer Verlag, April 2001.

[9] E.W. Felten and M.A. Schneider. Timing attacks on Web privacy. In Sushil Jajodia, editor, *7th ACM Conference on Computer and Communications Security*, pages 25–32, Athens, Greece, November 2000. ACM Press.

[10] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[11] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[12] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *8th International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212, Warsaw, Poland, 1997. Springer.

[13] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[14] N. Heintze and J.G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 1998.

[15] F. Henglein and D. Sands. A semantic model of binding times for safe partial evaluation. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982, pages 299–320. Springer-Verlag, 1995.

[16] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, July 2000.

[17] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of 29th ACM Symposium on Principles of Programming Languages, ACM Press*, Portland, Oregon, January 2002.

[18] J-J. Lévy. Some results on the join-calculus. In Martín Abadi and Takayasu Ito, editors, *Third International Symposium on Theoretical Aspects of Computer Software (Proceedings of TACS '97, Sendai, Japan)*, volume 1281 of *LNCS*. Springer, 1997.

[19] A.C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press.

[20] M. Odersky, C. Zenger, M. Zenger, and G. Chen. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999.

[21] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(4), 1997.

[22] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2002.

[23] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, Montréal, Canada, September 2000.

[24] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[25] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1575 of

Lecture Notes in Computer Science, pages 40–58. Springer-Verlag, 1999.

[26] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000. IEEE Computer Society Press.

[27] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *The 25thACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

Session II

Logical Approaches

(joint with VERIFY)

A Trustworthy Proof Checker

Andrew W. Appel*
Department of Computer Science
Princeton University
appel@cs.princeton.edu

Neophytos Michael*
Department of Computer Science
Princeton University
nmichael@cs.princeton.edu

Aaron Stump†
Computer Science Department
Stanford University
stump@stanford.edu

Roberto Virga*
Department of Computer Science
Princeton University
rvirga@cs.princeton.edu

Abstract

Proof-Carrying Code (PCC) and other applications in computer security require machine-checkable proofs of properties of machine-language programs. The main advantage of the PCC approach is that the amount of code that must be explicitly trusted is very small: it consists of the logic in which predicates and proofs are expressed, the safety predicate, and the proof checker. We have built a minimal-TCB checker, and we explain its design principles, and the representation issues of the logic, safety predicate, and safety proofs. We show that the trusted code in such a system can indeed be very small. In our current system the TCB is less than 2,700 lines of code (an order of magnitude smaller even than other PCC systems) which adds to our confidence of its correctness.

1 Introduction

Machine-verified proofs have applications in computer security, program verification, and the formalization of mathematics. We are particularly interested in security applications such as proof-carrying code, in which an untrusted program provider must send a proof that the machine-language program is safe to execute; or proof-carrying authorization, in which an untrusted client must send a proof that some desired server operation conforms to an authorization policy. The recipient doesn't trust the proof provider, so the recipient's proof checker is an essential component of the trusted computing base: a bug in the proof checker can be a security hole in the larger system. Therefore, the checker must be trustworthy: it must be small, simple, readable, and based on well-understood engineering and mathematical principles.

In contrast, theorem provers are often large and ugly, as required by the incompleteness results of Gödel and Turing: no prover of bounded size is sufficiently general, but one can always hack more features into the prover until it proves the desired class of theorems. It is difficult to fully trust such software, so some proving systems use technical means to ensure that buggy provers cannot produce invalid proofs: the abstract data type *theorem* of LCF [14], or the proof-witness objects of Coq [8] or Twelf [20]. With these means, only a small part of a large system must be examined and trusted.

How large is the proof checker that must be examined and trusted? To answer this question we have tried the experiment of constructing and measuring the *smallest possible* useful proof checker for some real application. Our checker receives, checks the safety of, and executes, proof-carrying code: machine code for the Sparc with an accompanying proof of safety. The proof is in higher-order logic represented in LF notation.

This checker would also be directly useful for proof-carrying authorization [3, 9], that is, checking proofs of authentication and permission according to some distributed policy.

A useful measure of the effort required to examine, understand, and trust a program is its size in (non-blank, non-comment) lines of source code. Although there may be much variation in effort and complexity per line of code, a crude quantitative measure is better than none. It is also necessary to count, or otherwise account for, any compiler, libraries, or supporting software used to execute the program. We address this issue explicitly by avoiding the use of libraries and by making the checker small enough so that it can be examined in machine language.

The *trusted computing base* (TCB) of a proof-carrying code system consists of all code that must be explicitly trusted as correct by the user of the system. In our case

*This research was supported in part by DARPA award F30602-99-1-0519.

†his research was supported by DARPA/Air Force contract F33615-00-C-1693 and NSF contract CCR-9806889.

the TCB consists of two pieces: first, the specification of the safety predicate in higher-order logic, and second, the proof checker, a small C program that checks proofs, loads, and executes safe programs.

In his investigation of Java-enabled browsers [11], Ed Felten found that the first-generation implementations averaged one security-relevant bug per 3,000 lines of source code [13]. These browsers, as mobile-code host platforms that depend on static checking for security, exemplify the kind of application for which proof-carrying code is well suited. Wang and Appel [7] measured the TCBs of various Java Virtual Machines at between 50,000 and 200,000 lines of code. The SpecialJ JVM [10] uses proof-carrying code to reduce the TCB to 36,000 lines.

In this work, we show how to reduce the size of the TCB to under 2,700 lines, and by basing those lines on a well understood logical framework, we have produced a checker which is small enough so that it can be manually verified; and as such it can be relied upon to accept only valid proofs. Since this small checker "knows" only about machine instructions, and nothing about the programming language being compiled and its type system, the semantic techniques for generating the proofs that the TCB will check can be involved and complex [2], but the checker doesn't.

2 The LF logical framework

For a proof checker to be simple and correct, it is helpful to use a well designed and well understood representation for logics, theorems, and proofs. We use the LF logical framework.

LF [15] provides a means for defining and presenting logics. The framework is general enough to represent a great number of logics of interest in mathematics and computer science (for instance: first-order, higher-order, intuitionistic, classical, modal, temporal, relevant and linear logics, and others). The framework is based on a general treatment of syntax, rules, and proofs by means of a typed first-order λ-calculus with dependent types. The LF type system has three levels of terms: objects, types, and kinds. Types classify objects and kinds classify families of types. The formal notion of definitional equality is taken to be $\beta\eta$-conversion.

A logical system is presented by a signature, which assigns types and kinds to a finite set of constants that represent its syntax, its judgments, and its rule schemes. The LF type system ensures that object-logic terms are well formed. At the proof level, the system is based on the *judgments-as-types* principle: judgments are represented as types, and proofs are represented as terms whose type is the representation of the theorem they prove. Thus, there is a correspondence between type-checked terms and theorems of the object logic. In this way proof checking of the object logic is reduced to type checking of the LF terms.

For developing our proofs, we use Twelf [20], an implementation of LF by Frank Pfenning and his students. Twelf is a sophisticated system with many useful features: in addition to an LF type checker, it contains a type-reconstruction algorithm that permits users to omit many explicit parameters, a proof-search algorithm, constraint regimes (e.g., linear programming over the exact rational numbers), mode analysis of parameters, a meta-theorem prover, a pretty-printer, a module system, a configuration system, an interactive Emacs mode, and more. We have found many of these features useful in proof development, but Twelf is certainly not a minimal proof checker. However, since Twelf does construct explicit proof objects internally, we can extract these objects to send to our minimal checker.

In LF one declares the operators, axioms, and inference rules of an *object logic* as constructors. For example, we can declare a fragment of first-order logic with the type `form` for formulas and a dependent type constructor `pf` for proofs, so that for any formula `A`, the type `pf(A)` contains values that are proofs of `A`. Then, we can declare an "implies" constructor `imp` (infix, so it appears between its arguments), so that if `A` and `B` are formulas then so is `A imp B`. Finally, we can define introduction and elimination rules for `imp`.

```
form : type.
pf   : form -> type.
imp  : form -> form -> form. %in-
fix right 10 imp.
imp_i: (pf A -> pf B) -> pf (A imp B).
imp_e: pf (A imp B) -> pf A -> pf B.
```

All the above are defined as constructors. In general, constructors have the form $name : \tau$ and declare that $name$ is a value of type τ.

It is easy to declare inconsistent object-logic constructors. For example, `invalid: pf A` is a constructor that acts as a proof of any formula, so using it we could easily prove the false proposition:

```
logic_inconsistent : pf (false) = invalid.
```

So the object logic should be designed carefully and must be trusted.

Once the object logic is defined, theorems can be proved. We can prove for instance that implication is transitive:

```
imp_trans:
  pf (A imp B) -> pf (B imp C) -
> pf (A imp C) =
   [p1: pf (A imp B)][p2: pf (B imp C)]
    imp_i [p3: pf A] imp_e p2 (imp_e p1 p3).
```

In general, definitions (including predicates and theorems) have the form $name : \tau = exp$, which means that $name$ is now to stand for the value exp whose type is τ. In this example, the exp is a function with formal parameters `p1` and `p2`, and with body `imp_i [p3] imp_e p2 (imp_e p1 p3)`.

Definitions need not be trusted, because the type-checker can verify whether exp does have type τ. In general, if a proof checker is to check the proof P of theorem T in a logic $\mathcal{L}$, then the constructors (operators and axioms) of $\mathcal{L}$ must be given to the checker in a trusted way (i.e., the adversary must not be free to install inconsistent axioms). The statement of T must also be trusted (i.e., the adversary must not be free to substitute an irrelevant or vacuous theorem). The adversary provides only the proof P, and then the checker does the proof checking (i.e., it type-checks in the LF type system the definition $t : T = P$, for some arbitrary name t).

3 Application: Proof-carrying code

Our checker is intended to serve a purpose: to check safety theorems about machine-language programs. It is important to include application-specific portions of the checker in our measurements to ensure that we have adequately addressed all issues relating to interfacing to the real world.

The most important real-world-interface issue is, "is the proved theorem meaningful?" An accurate checker does no good if it checks the wrong theorem. As we will explain, the specification of the safety theorem is larger than all the other components of our checker combined!

Given a machine-language program P, that is, a sequence of integers that code for machine instructions (on the Sparc, in our case), the theorem is, "when run on the Sparc, P never executes an illegal instruction, nor reads or writes from memory outside a given range of addresses." To formalize this theorem it is necessary to formalize a description of instruction execution on the Sparc processor. We do this in higher-order logic augmented with arithmetic.

In our model [16], a machine state (r, m) comprises a *register bank* (r), and a *memory* (m), each of which is a function from integers (register numbers and addresses) to integers (contents). Every register of the instruction-set architecture (ISA) must be assigned a number in the register bank: the general registers, the floating-point registers, the condition codes, and the program counter. Where the ISA does not specify a number (such as for the PC) or when the numbers for two registers conflict (such as for the floating point and integer registers) we use an arbitrary unused index.

A single step of the machine is the execution of one instruction. We can specify instruction execution by giving a step relation $(r, m) \mapsto (r', m')$ that maps the prior state (r, m) to the next state (r', m') that holds after the execution of the machine instruction.

For example, to describe the "add" instruction $r_1 \leftarrow r_2 + r_3$ we might start by writing,

$$(r, m) \mapsto (r', m') \equiv r'(1) = r(2) + r(3) \\ \wedge (\forall x \neq 1.\, r'(x) = r(x)) \wedge m' = m$$

In fact, we can parameterize the above on the three registers involved and define $\mathrm{add}(i, j, k)$ as the following predicate on four arguments (r, m, r', m'):

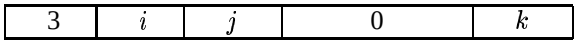
$$\mathrm{add}(i, j, k) \equiv \\ \lambda r, m, r', m'.\ r'(i) = r(j) + r(k) \\ \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

Similarly, for the "load" instruction $r_i \leftarrow m[r_j + c]$ we define its semantics to be the predicate:

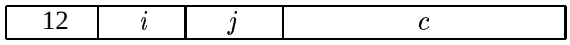
$$\mathrm{load}(i, j, c) \equiv \\ \lambda r, m, r', m'.\ r'(i) = m(r(j) + c) \\ \wedge (\forall x \neq i.\, r'(x) = r(x)) \wedge m' = m$$

To enforce memory safety policies, we will modify the definition of $\mathrm{load}(i, j, c)$ to require that the loaded address is legal [2], but we omit those details here.

But we must also take into account instruction fetch and decode. Suppose, for example, that the "add" instruction is encoded as a 32-bit word, containing a 6-bit field with opcode 3 denoting *add*, a 5-bit field denoting the destination register i, and 5-bit fields denoting the source registers j, k:



The "load" instruction might be encoded as:



Then we can say that some number w decodes to an instruction *instr* iff,

$$\mathrm{decode}(w, \mathit{instr}) \equiv \\ (\exists i, j, k. \\ 0 \leq i < 2^5 \ \wedge\ 0 \leq j < 2^5 \ \wedge\ 0 \leq k < 2^5 \ \wedge \\ w = 3 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + k \cdot 2^0 \ \wedge \\ \mathit{instr} = \mathrm{add}(i, j, k)) \\ \vee\ (\exists i, j, c. \\ 0 \leq i < 2^5 \ \wedge\ 0 \leq j < 2^5 \ \wedge\ 0 \leq c < 2^{16} \ \wedge \\ w = 12 \cdot 2^{26} + i \cdot 2^{21} + j \cdot 2^{16} + c \cdot 2^0 \ \wedge \\ \mathit{instr} = \mathrm{load}(i, j, c)) \\ \vee\ \ldots$$

with the ellipsis denoting the many other instructions of the machine, which must also be specified in this formula.

We have shown [16] how to scale this idea up to the instruction set of a real machine. Real machines have large but semi-regular instruction sets; instead of a single global disjunction, the decode relation can be factored into operands, addressing modes, and so on. Real machines don't use integer arithmetic, they use modular arithmetic, which can itself be specified in our higher-order logic. Some real machines have multiple program counters (e.g., Sparc) or variable-length instructions (e.g., Pentium), and these can also be accommodated.

Our description of the decode relation is heavily factored by higher-order predicates (this would not be possible without higher-order logic). We have specified the execution behavior of a large subset of the Sparc architecture, and we have built a prototype proof-generating compiler that targets that subset. For proof-carrying code, it is sufficient to specify a subset of the machine architecture; any unspecified instruction will be treated by the safety policy as illegal. While this may be inconvenient for compilers that want to generate that instruction, it does ensure that safety cannot be compromised.

4 Specifying safety

Our step relation $(r, m) \mapsto (r', m')$ is deliberately partial; some states have no successor state. In these states the program counter $r(\text{PC})$ points to an illegal instruction. Using this partial step relation, we can define safety. A safe program is one that will never execute an illegal instruction; that is, a given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state:

$$\text{safe-state}(r, m) \equiv \forall r', m'.\ (r, m) \mapsto^* (r', m') \Rightarrow \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

A program is just a sequence of integers (each representing a machine instruction); we say that a program p is loaded at a location l in memory m if

$$\text{loaded}(p, m, l) \equiv \forall i \in \text{dom}(p).\ m(i + l) = p(i)$$

Finally (assuming that programs are written in position-independent code), a program is *safe* if, no matter where we load it in memory, we get a safe state:

$$\text{safe}(p) \equiv \forall r, m, l.\ \text{loaded}(p, m, l) \wedge r(\text{PC}) = l \Rightarrow \text{safe-state}(r, m)$$

Let `;` be a "cons" operator for sequences of integers (easily definable in HOL); then for some program `8420; 2837; 2938; 2384; nil` the safety theorem is simply:

```
safe (8420; 2837; 2938; 2384; nil)
```

and, given a proof P, the LF definition that must be type-checked is:

```
t: pf(safe(8420; 2837; 2938; 2384; nil)) = P.
```

Though we wrote in section 2 that definitions need not be trusted because they can be type-checked, this is not strictly true. Any definition used in the statement of the theorem must be trusted, because the wrong definition will lead to the proof of the wrong theorem. Thus, all the definitions leading up to the definition of `safe` (including `add`, `load`, `safe-state`, `step`, etc.) must be part of the trusted checker. Since we have approximately 1,600 lines of such definitions, and they are a component of our "minimal" checker, one of the most important issues we faced is the representation of these definitions; we discuss this in Section 7.

On the other hand, a large proof will contain hundreds of internal definitions. These are predicates and internal lemmas of the proof (not of the statement of the theorem), and are at the discretion of the proof provider. Since each is type checked by the checker before it is used in further definitions and proofs, they don't need to be trusted.

In the table below we show the various pieces needed for the specification of the safety theorem in our logic. Every piece in this table is part of the TCB. The first two lines show the size of the logical and arithmetic connectives (in which theorems are specified) as well as the size of the logical and arithmetic axioms (using which theorems are proved). The Sparc specification has two components, a "syntactic" part (the decode relation) and a semantic part (the definitions of *add*, *load*, etc.); these are shown in the next two lines. The size of the safety predicate is shown last.

Safety Specification	*Lines*	*Definitions*
Logic	135	61
Arithmetic	160	94
Machine Syntax	460	334
Machine Semantics	1,005	692
Safety Predicate	105	25
Total	1,865	1,206

From this point on we will refer to everything in the table as the *safety specification* or simply the *specification*.

5 Eliminating redundancy

Typically an LF signature will contain much redundant information. Consider for example the rule for `imp` introduction presented previously; in fully explicit form, their representation in LF is as follows:

```
imp_i : {A : form}{B : form}
        (pf A -> pf B) -> pf (A imp B).
```

The fact that both `A` and `B` are formulas can be easily inferred by the fact they are given as arguments to the constructor `imp`, which has been previously declared as an operator over formulas.

On the one hand, eliminating redundancy from the representation of proofs benefits both proof size and type-checking time. On the other hand, it requires performing term reconstruction, and thus it may dramatically increase the complexity of type checking, driving us away from our goal of building a minimal checker.

Twelf deals with redundancy by allowing the user to declare some parameters as *implicit*. More precisely, all variables which are not quantified in the declaration are

automatically assumed implicit. Whenever an operator is used, Twelf's term reconstruction will try to determine the correct substitution for all its implicit arguments. For example, in type-checking the lemma

```
imp_refl: pf (A imp A) = imp_i ([p : pf A] p).
```

Twelf will automatically reconstruct the two implicit arguments of `imp_i` to be both equal to `A`.

While Twelf's notion of implicit arguments is effective in eliminating most of the redundancy, type reconstruction adds considerable complexity to the system. Another drawback of Twelf's type reconstruction is its reliance on higher-order unification, which is undecidable. Because of this, type checking of some valid proofs may fail.

Since full type reconstruction is too complex to use in a trusted checker, one might think of sending fully explicit LF proof terms; but a fully explicit proof in Twelf syntax can be exponentially larger than its implicit representation. To avoid these problems, Necula's LF$_i$ [18] uses partial type reconstruction and a simple algorithm to determine which of the arguments can be made implicit. Implicit arguments are omitted in the representation, and replaced by placeholders. Oracle-based checking [19] reduces the proof size even further by allowing the erasure of subterms whose reconstruction is not uniquely determined. Specifically, in cases when the reconstruction of a subterm is not unique, but there is a finite (and usually small) list of candidates, it stores an oracle for the right candidate number instead of storing the entire subterm.

These techniques use clever syntactic representations of proofs that minimize proof size; checking these representations is not as complex as full Twelf-style type reconstruction, but is still more complex than is appropriate for a minimal proof checker. We are willing to tolerate somewhat larger proofs in exchange for a really simple checking algorithm. Instead of using a syntactic representation of proofs, we avoid the need for the checker to parse proofs by using a data-structure reprentation. However, we still need to avoid exponential blowups, so we reduce redundancy by structure sharing. Therefore, we represent and transmit proofs as LF terms in the form of directed acyclic graphs (DAGs), with structure sharing of common subexpressions to avoid exponential blowup.

A node in the DAG can be one of ten possible types: one for kinds, five for ordinary LF terms, and four for arithmetic expressions. Each node may store up to three integers, `arg1`, `arg2`, and `type`. This last one, if present, will always point to the sub-DAG representing the type of the expression.

	arg1	arg2	type	
n	U	U	U	kind
c	U	U	M	constant
v	M	M	M	variable
a	M	M	O	application
p	M	M	O	product
l	M	M	O	abstraction
#	M	U	O	number
+	M	M	O	addition proof object
*	M	M	O	mult proof object
/	M	M	O	div proof object

M = mandatory, O = optional, U = unused

The content of `arg1` and `arg2` is used in different ways for different node types. For all nodes representing arithmetic expressions ('#', '+', '*', and '/'), they contain integer values. For products and abstractions ('p' and 'l'), `arg1` points to the bound variable, and `arg2` to the term where the binding takes place. For variable nodes ('v'), they are used to make sure that the variable always occurs within the scope of a quantifier. For application nodes ('a'), they point to the function and its argument, respectively. Finally, constant declaration nodes ('c'), and kind declaration nodes ('n') use neither.

For a concrete example, consider the LF signature:

```
form : type.
pf   : form -> type.
imp  : form -> form -> form.
```

We present below the DAG representation of this signature. We "flattened" the DAG into a numbered list, and, for clarity, we also added a comment on the right showing the corresponding LF term.

```
 1| n 0 0 0   ; type Kind
 2| c 0 0 1   ; form: type
 3| v 0 0 2   ; x: form
 4| p 3 1 0   ; {x: form} type
 5| c 0 0 4   ; pf: {x: form} type
 6| v 0 0 2   ; y: form
 7| p 6 2 0   ; {y: form} form
 8| v 0 0 2   ; x: form
 9| p 8 7 0   ; {x: form}{y: form} form
10| c 0 0 9   ; imp: {x: form}{y: form} form
```

6 Dealing with arithmetic

Since our proofs reason about encodings of machine instructions (opcode calculations) and integer values manipulated by programs, the problem of representing arithmetic within our system is a critical one. A purely logical representation based on 0, successor and predecessor is not suitable to us, since it would cause proof size to explode.

The latest releases of Twelf offer extensions that deal natively with infinite-precision integers and rationals. While these extensions are very powerful and convenient to use, they offer far more than we need, and because of their generality they have a very complex implementation (the

rational extension alone is 1,950 lines of Standard ML). What we would like for our checker is an extension built in the same spirit as those, but much simpler and lighter.

We require two properties from such an extension:

1. LF terms for all the numbers we use; moreover, the size of the LF term for n should be constant and independent of n.

2. Proof objects for single-operation arithmetic facts such as "$10 + 2 = 12$"; again, we require that such proof objects have constant size.

Our arithmetic extensions to the checker are the smallest and simplest ones to satisfy (1) and (2) above. We add the `word32` type to the TCB, (representing integers in the range $[0, 2^{32} - 1]$) as well as the following axioms:

```
+ : word32 -> word32 -> word32 -> type.
* : word32 -> word32 -> word32 -> type.
/ : word32 -> word32 -> word32 -> type.
```

We also modify the checker to accept arithmetic terms such as:

```
456+25 : + 456 25 481.
32*4   : * 32 4 128.
```

This extension does not modify in any way the standard LF type checking: we could have obtained the same result (although much more inefficiently) if we added all these constants to the trusted LF signature by hand. However, granting them special treatment allowed us to save literally millions of lines in the axioms in exchange for an extra 55 lines in the checker.

To embed and use these new constants in our object logic, we also declare:

```
c: word32 -> tm num.

eval_plus: + A B C ->
   pf (eq (plus (c A) (c B)) (c C)).

eval_times: * A B C ->
   pf (eq (times (c A) (c B)) (c C)).

eval_div: / M N Q ->
   pf ((geq (c M) (times (c N) (c Q))) and
       (not (geq (c M) (times (c N)
                         (plus one (c Q)))))).
```

This embedding from `word32` to numbers in our object logic is not surjective. Numbers in our object logic are still unbounded; `word32` merely provides us with handy names for the ones used most often.

With this "glue" to connect object logic to meta logic, numbers and proofs of elementary arithmetic properties, are just terms of size two.

7 Representing axioms and trusted definitions

Since we can represent axioms, theorems, and proofs as DAGs, it might seem that we need neither a parser nor a pretty-printer in our minimal checker. In principle, we could provide our checker with an initial trusted DAG representing the axioms and the theorem to be proved, and then it could receive and check an untrusted DAG representing the proof. The trusted DAG could be represented in the C language as an initialized array of graph nodes.

This might work if we had a very small number of axioms and trusted definitions, and if the statement of the theorem to be proved were very small. We would have to read and trust the initialized-array statements in C, and understand their correspondence to the axioms (etc.) as we would write them in LF notation. For a sufficiently small DAG, this might be simpler than reading and trusting a parser for LF notation.

However, even a small set of operators and axioms (especially once the axioms of arithmetic are included) requires hundreds of graph nodes. In addition, as explained in section 4, our trusted definitions include the machine-instruction step relation of the Sparc processor. These 1,865 lines of Twelf expand to 22,270 DAG nodes. Clearly it is impossible for a human to directly read and trust a graph that large.

Therefore, we require a parser or pretty-printer in the minimal checker; we choose to use a parser. Our C program will parse the 1,865 lines of axioms and trusted definitions, translating the LF notation into DAG nodes. The axioms and definitions are also part of the C program: they are a constant string to which the parser is applied on startup.

This parser is 428 lines of C code; adding these lines to the minimal checker means our minimal checker can use 1,865 lines of LF instead of 22,270 lines of graph-node initializers, clearly a good tradeoff. Our parser accepts valid LF expressions, written in the same syntax used by Twelf. For more details see the full version of the paper [6].

7.1 Encoding higher-order logic in LF

Our encoding of higher-order logic in LF follows that of Harper et al. [15] and is shown in figure 1. The *constructors* generate the syntax of the object logic and the *axioms* generate its proofs. A meta-logical type is `type` and an object-logic type is `tp`. Object-logic types are constructed from `form` (the type of formulas), `num` (the type of integers), and the `arrow` constructor. The LF term `tm` maps an object type to a meta type, so an object-level term of type `T` has type `(tm T)` in the meta logic.

Abstraction in the object logic is expressed by the `lam` term. The term `(lam [x] (F x))` is the object-logic function that maps `x` to `(F x)`. Application for such

Logic Constructors

```
tp        :  type.
tm        :  tp -> type.
form      :  tp.
arrow     :  tp -> tp -> tp.
pf        :  tm form -> type.
lam       :  (tm T1 -> tm T2) -> tm (T1 arrow T2).
@         :  tm (T1 arrow T2) -> tm T1 -> tm T2.
forall    :  (tm T -> tm form) -> tm form.
imp       :  tm form -> tm form -> tm form.
```

Logic Axioms

```
beta_e    :  pf (P ((lam F) @ X)) -> pf (P (F X)).
beta_i    :  pf (P (F X)) -> pf (P (lam F) @ X).
imp_i     :  (pf A -> pf B) -> pf (A imp B).
imp_e     :  pf (A imp B) -> pf A -> pf B.
forall_i  :  ({X : tm T} pf (A X)) -> pf (forall A).
forall_e  :  pf (forall A) -> {X : tm T} pf (A X).
```

Figure 1: Higher-Order Logic in Twelf

lambda terms is expressed via the `@` operator. The quantifier `forall` is defined to take as input a meta-level (LF) function of type `(tm T -> tm form)` and produce a `tm form`. The use of the LF functions here makes it easy to perform substitution when a proof of `forall` needs to be discharged, since equality in LF is just $\beta\eta$-conversion.

Notice that most of the standard logical connectives are absent from figure 1. This is because we can produce them as definitions from the constructors we already have. For instance, conjunction can be defined as follows:

```
and = [A][B] forall [C] (A imp B imp C) imp C.
```

It is easy to see that the above formula is equivalent to the standard definition of `and`. We can likewise define introduction and elimination rules for all such logic constructors. These rules are proven as lemmas and need not be trusted. Object-level equality[1] is also easy to define:

```
eq : tm T -> tm T -> tm form =
 [A][B] forall [P] P @ B imp P @ A.
```

This states that objects `A` and `B` are considered equal iff any predicate `P` that holds on `B` also holds on `A`.

Terms of type `pf A` are terms representing proofs of object formula `A`. Such terms are constructed using the axioms of figure 1. Axioms `beta_i` and `beta_e` are used to prove β-equivalence in the object logic, `imp_i` and `imp_e` transform a meta-level proof function to the object level and vice-versa, and finally, `forall_i` and `forall_e` introduce and eliminate the universal quantifier.

7.2 "Polymorphic" programming in Twelf

ML-style implicit polymorphism allows one to write a function usable at many different argument types, and ML-style type inference does this with a low syntactic overhead. We are writing proofs, not programs, but we would still like to have polymorphic predicates and polymorphic lemmas. LF is not polymorphic in the ML sense, but Harper et al. [15] show how to use LF's dependent type system to get the effect and (most of) the convenience of implicit parametric polymorphism with an encoding trick, which we will illustrate with an example.

Suppose we wish to write the lemma `congr` that would allow us to substitute equals for equals:

```
congr : {H : type -> tm form}
 pf (eq X Z) -> pf (H Z) -> pf (H X) = ...
```

The lemma states that for any predicate `H`, if `H` holds on `Z` and `Z` = `X` then `H` also holds on `X`. Unfortunately this is ill-typed in LF since LF does not allow polymorphism. Fortunately though, there is way to get polymorphism at the object level. We rewrite `congr` as:

```
congr : {H : tm T -> tm form}
 pf (eq X Z) -> pf (H Z) -> pf (H X) = ...
```

and this is now acceptable to Twelf. Function `H` now judges objects of meta-type `(tm T)` for any object-level type `T`, and so `congr` is now "polymorphic" in `T`. We can apply it on any object-level type, such as `num`, `form`, `num arrow form`, etc. This solution is general enough to

[1] The equality predicate `eq` is polymorphic in `T`. Objects `A` and `B` have object type `T` and so they could be `num`s, `form`s or even object level functions (`arrow` types). The object type `T` is implicit in the sense that when we use the `eq` predicate we do not have to specify it; Twelf can automatically infer it. So internally, the meta-level type of `eq` is not what we have specified above but the following:

```
{T : tp} tm T -> tm T -> tm form.
```

We will have more to say about this in section 7.2.

allow us to express any polymorphic term or lemma with ease. Axioms `forall_i` and `forall_e` in figure 1 are likewise polymorphic in `T`.

7.3 How to write explicit Twelf

In the definition of lemma `congr` above, we have left out many explicit parameters since Twelf's type-reconstruction algorithm can infer them. The explicit version of the LF term `congr` is:

```
congr : {T : tp}{X : tm T}{Z : tm T}
        {H : tm T -> tm form}
 pf (_eq T X Z) -> pf (H Z) -> pf (H X) = ...
```

(here we also make use of the explicit version of the equality predicate `_eq`). Type reconstruction in Twelf is extremely useful, especially in a large system like ours, where literally hundreds of definitions and lemmas have to be stated and proved.

Our safety specification was originally written to take advantage of Twelf's ability to infer missing arguments. Before proof checking can begin, this specification needs to be fed to our proof checker. In choosing then what would be in our TCB we had to decide between the following alternatives:

1. Keep the implicitly-typed specification in the TCB and run it through Twelf to produce an explicit version (with no missing arguments or types). This explicit version would be fed to our proof checker. This approach allows the specification to remain in the implicit style. Also our proof checker would remain simple (with no type reconstruction/inference capabilities) but unfortunately we now have to add to the TCB Twelf's type-reconstruction and unification algorithms, which are about 5,000 lines of ML code.

2. Run the implicitly typed specification through Twelf to get an explicit version. Now instead of trusting the implicit specification and Twelf's type-reconstruction algorithms, we keep them out of the TCB and proceed to manually verify the explicit version. This approach also keeps the checker simple. Unfortunately the explicit specification produced by Twelf explodes in size from 1,700 to 11,000 lines, and thus the code that needs to be verified is huge. The TCB would grow by a lot.

3. Rewrite the trusted definitions in an explicit style. Now we do not need type reconstruction in the TCB (the problem of choice 1), and if the rewrite from the implicit to the explicit style can avoid the size explosion (the problem of choice 2), then we have achieved the best of both worlds.

Only choice 3 was consistent with our goal of a small TCB so we rewrote the trusted definitions in an explicit style while managing to avoid the size explosion. The new safety specification is only 1,865 lines of explicitly-typed Twelf. It contains no terms with implicit arguments and so we do not need a type-reconstruction/type-inference algorithm in the proof checker. The rewrite solves the problem while maintaining the succinctness and brevity of the original TCB, the penalty of the explicit style being an increase in size of 124 lines. The remainder of this section explains the problem in detail and the method we used to bypass it.

To see why there is such an enormous difference in size (1,700 lines vs 11,000) between the implicit specification and its explicit representation generated by Twelf's type-reconstruction algorithm, consider the following example. Let `F` be a two-argument object-level predicate `F : tm (num arrow num arrow form)` (typical case when describing unary operators in an instruction set). When such a predicate is applied, as in `(F @ X @ Y)`, Twelf has to infer the implicit arguments to the two instances of operator `@`. The explicit representation of the application then becomes:

```
@ num form (@ num (num arrow form) F X) Y
```

It is easy to see how the explicit representation explodes in size for terms of higher order. Since the use of higher-order terms was essential in achieving maximal factoring in the machine descriptions [16], the size of the explicit representation quickly becomes unmanageable.

Here is another more concrete example from the `decode` relation of section 3. This one shows how the abstraction operator `lam` suffers from the same problem. The predicate below (given in implicit form) is used in specifying the syntax of all Sparc instructions of two arguments.

```
fld2 = lam6 [f0][f1][p_pi][icons][ins][w]
 p_pi @ w and
 exists2 [g0][g1] (f0 @ g0 && f1 @ g1) @ w and
                  eq ins (icons @ g0 @ g1).
```

Predicates `f0` and `f1` specify the input and the output registers, `p_pi` decides the instruction opcode, `icons` is the instruction constructor, `ins` is the instruction we are decoding, and `w` is the machine-code word. In explicit form this turns into what we see on the left-hand side of figure 2 – an explicitly typed definition 16 lines long.

The way around this problem is the following: We avoid using object-logic predicates whenever possible. This way we need not specify the types on which object-logic application and abstraction are used. For example, the `fld2` predicate above now becomes what we see on the right-hand side of figure 2. This new predicate has shrunk in size by more than half.

Sometimes moving predicates to the meta-logic is not possible. For instance, we represent *instructions* as predicates from machine states to machine states (see section 3). Such predicates must be in the object logic since we need to be able to use them in quantifiers (`exists [ins : tm instr] ...`). Thus, we face the

Object Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 lam6 (arrow T1 (arrow T2 form))
      (arrow T3 (arrow T2 form))
      (arrow T2 form)
      (arrow T1 (arrow T3 T4))
      T4 T2 form
  [f0][f1][p_pi][icons][ins][w]
  (@ T2 form p_pi w) and
  (exists2 T1 T3 [g0:tm T1] [g1:tm T3]
    (@ T2 form
     (&& T2 (@ T1 (arrow T2 form) f0 g0)
        (@ T3 (arrow T2 form) f1 g1)) w) and
    (eq T4 ins (@ T3 T4 (@ T1 (arrow T3 T4)
                              icons g0) g1))).
```

Meta Logic Abstraction/Application

```
fld2 = [T1:tp][T2:tp][T3:tp][T4:tp]
 [f0][f1][p_pi][icons][ins][w]
 (p_pi w) and
 (exists2 [g0:tm T1][g1:tm T3]
  (f0 g0 && f1 g1) w) and
  (eq ins (icons g0 g1)).
```

Figure 2: Abstraction & Application in the Object versus Meta Logic.

problem of having to supply all the implicit types when defining and applying such predicates. But since these types are always fixed we can factor the partial applications and avoid the repetition. For example, when defining some Sparc machine instruction as in:

```
i_anyInstr = lam2 [rs : tnum][rd : tnum]
 lam4 registers memory registers memory form
  [r : tregs][m : tmem][r' : tregs][m' : tmem]
   ...
```

we define the predicate `instr_lam` as:

```
instr_lam = lam4 registers memory
                 registers memory form.
```

and then use it in defining each of the 250 or so Sparc instructions as below:

```
i_anyInstr = [rs : tnum][rd : tnum]
 instr_lam [r : tregs][m : tmem]
           [r' : tregs][m' : tmem] ...
```

This technique turns out to be very effective because our machine syntax and semantics specifications were highly factored to begin with [16].

So by moving to the meta logic and by clever factoring we have moved the TCB from implicit to explicit style with only a minor increase in size. Now we don't have to trust a complicated type-reconstruction/type-inference algorithm. What we feed to our proof checker is precisely the set of axioms we explicitly trust.

7.4 The implicit layer

When we are building proofs, we still wish to use the implicit style because of its brevity and convenience. For this reason we have built an implicit layer on top of our explicit TCB. This allows us to write proofs and definitions in the implicit style and LF's $\beta\eta$-conversion takes care of establishing meta-level term equality. For instance, consider the object-logic application operator _@ given below:

```
_@: {T1 : tp}{T2 : tp} tm (T1 arrow T2) ->
                          tm T1 -> tm T2.
```

In the implicit layer we now define a corresponding application operator @ in terms of _@ as follows:

```
@: tm (T1 arrow T2) -> tm T1 -> tm T2 =
    _@ T1 T2.
```

In this term the type variables `T1` and `T2` are implicit and need not be specified when @ is used. Because @ is a definition used only in proofs (not in the statement of the safety predicate), it does not have to trusted.

8 The proof checker

The total number of lines of code that form our checker is 2,668. Of these, 1,865 are used to represent the LF signature containing the core axioms and definition, which is stored as a static constant string. The remaining 803 lines of C source code, can be broken down as follows:

Component	*Lines*
Error messaging	14
Input/Output	29
Parser	428
DAG creation and manipulation	111
Type checking and term equality	167
Main program	54
Total	803

We make no use of libraries in any of the components above. Libraries often have bugs, and by avoiding their use we eliminate the possibility that some adversary may exploit one of these bugs to disable or subvert proof checking. However, we do make use of two POSIX calls: `read`, to read the program and proof, and `_exit`, to quit if the proof is invalid. This seems to be the minimum possible use of external libraries.

8.1 Trusting the C compiler

We hasten to point out that these 803 lines of C need to be compiled by a C compiler, and so it would appear that this compiler would need to be included in our TCB. The C compiler could have bugs that may potentially be exploited by an adversary to circumvent proof checking. More dangerously perhaps, the C compiler could have been written by the adversary so while compiling our checker, it could insert a Thompson-style [22] Trojan horse into the executable of the checker.

All proof verification systems suffer from this problem. One solution (as suggested by Pollack [21]) is that of independent checking: write the proof checker in a widely used programming language and then use different compilers of that language to compile the checker. Then, run all your checkers on the proof in question. This is similar to the way mathematical proofs are "verified" as such by mathematicians today.

The small size of our checker suggests another solution. Given enough time one may read the output of the C compiler (assembly language or machine code) and verify that this output faithfully implements the C program given to the compiler. Such an examination would be tedious but it is not out of the question for a C program the size of our checker (3900 Sparc instructions, as compiled), and it could be carried out if such a high level of assurance was necessary. Such an investigation would certainly uncover Thompson-style Trojan horses inserted by a malicious compiler. This approach would not be feasible for the JVMs mentioned in the introduction; they are simply too big.

8.2 Proof-checking measurements

In order to test the proof checker, and measure its performance, we wrote a small Standard ML program that converts Twelf internal data structures into DAG format, and dumps the output of this conversion to a file, ready for consumption by the checker.

We performed our measurements on a sample proof of nontrivial size, that proves a substantial lemma that will be used in proofs of real Sparc programs. (We have not yet built a full lemma base and prover that would allow us to test full safety proofs.) In its original formulation, our sample proof is 6,367 lines of Twelf, and makes extensive use of implicit arguments. Converted to its fully explicit form, its size expands to 49,809 lines. Its DAG representation consists of 177,425 nodes.

Checking the sample proof consists of the four steps: parsing the TCB, loading the proof from disk, checking the DAG for well-formedness, and type-checking the proof itself. The first three steps take less than a second to complete, while the last step takes 79.94 seconds.

The measurements above were made on a 1 GHz Pentium III PC with 256MB of memory. During type checking of this proof the number of temporary nodes generated is 1,115,768. Most of the time during type-checking is spent in performing substitutions. All the lemmas and definitions we use in our proofs are closed expressions, and therefore they do not need to be traversed when substituting for a variable. We are currently working on an optimization that will allow our checker to keep track of closed subexpressions and to avoid their traversal during substitution. We believe this optimization can be achieved without a significant increase in the size of the checker, and it will allow us to drastically reduce type-checking time.

9 Future work

The DAG representation of proofs is quite large, and we would like to do better. One approach would be to compress the DAGs in some way; another approach is to use a compressed form of the LF syntactic notation. However, we believe that the most promising approach is neither of these.

Our proofs of program safety are structured as follows: first we prove (by hand, and check by machine) many structural lemmas about machine instructions and semantics of types [4, 5, 1]. Then we use these lemmas to prove (by hand, as derived lemmas) the rules of a low-level typed assembly language (TAL). Our TAL has several important properties:

1. Each TAL operator corresponds to exactly 0 or 1 machine instructions (0-instruction operators are coercions that serve as hints to the TAL typechecker and do nothing at runtime).

2. The TAL rules prescribe Sparc instruction encodings as well as the more conventional [17] register names, types, and so on.

3. The TAL typing rules are syntax-directed, so typechecking a TAL expression is decidable by a simple tree-walk without backtracking.

4. The TAL rules can be expressed as a set of Horn clauses.

Although we use higher-order logic to state and prove lemmas leading up to the proofs of the TAL typing rules, and we use higher-order logic in the proofs of the TAL rules, we take care to make all the statements of the TAL rules first-order Horn clauses. Consider a clause such as: `head :- goal1 , goal2 , goal3`. In LF (using our object logic) we could express this as a lemma:

```
n : pf (goal3) -> pf (goal2) ->
     pf (goal1) -> pf (head) = proof.
```

Inside *proof* there may be higher-order abstract syntax, quantification, and so on, but the `goals` are all Prolog-style. The name `n` identifies the clause.

Our compiler produces Sparc machine code using a series of typed intermediate languages, the last of which is our TAL. Our prover constructs the safety proof for the Sparc program by "executing" the TAL Horn clauses as a logic program, using the TAL expression as input data. The proof is then a tree of clause names, corresponding to the TAL typing derivation.

We can make a checker that takes smaller proofs by just implementing a simple Prolog interpreter (without backtracking, since TAL is syntax-directed). But then we would need to trust the Prolog interpreter and the Prolog program itself (all the TAL rules). This is similar to what Necula [18] and Morrisett et al. [17] do. The problem is that in a full-scale system, the TAL comprises about a thousand fairly complex rules. Necula and Morrisett have given informal (i.e., mathematical) proofs of the soundness of their type systems for prototype languages, but no machine-checked proof, and no proof of a full-scale system.

The solution, we believe, is to use the technology we have described in this paper to check the derivations of the TAL rules from the logic axioms and the Sparc specification. Then, we can add a simple (non-backtracking) Prolog interpreter to our minimal checker, which will no longer be minimal: we estimate that this interpreter will add 200–300 lines of C code.

The proof producer (adversary) will first send to our checker, as a DAG, the definitions of Horn clauses for the TAL rules, which will be LF-typechecked. Then, the "proofs" sent for machine-language programs will be in the form of TAL expressions, which are much smaller than the proof DAGs we measured in section 8.

A further useful extension would be to implement oracle-based checking [19]. In this scheme, a stream of "oracle bits" guides the application of a set of Horn clauses, so that it would not be necessary to send the TAL expression – it would be re-derived by consulting the oracle. This would probably give the most concise safety proofs for machine-language programs, and the implementation of the oracle-stream decoder would not be too large. Again, in this solution the Horn clauses are first checked (using a proof DAG), and then they can be used for checking many successive TAL programs.

Although this approach seems very specific to our application in proof-carrying code, it probably applies in other domains as well. Our semantic approach to distributed authentication frameworks [3] takes the form of axioms in higher-order logic, which are then used to prove (as derived lemmas) first-order protocol-specific rules. While in that work we did not structure those rules as Horn clauses, more recent work in distributed authentication [12] does express security policies as sets of Horn clauses. By combining the approaches, we could have our checker first verify the soundness of a set of rules (using a DAG of higher-order logic) and then interpret these rules as a Prolog program.

10 Conclusion

Proof-carrying code has a number of technical advantages over other approaches to the security problem of mobile code. We feel that the most important of these is the fact that the trusted code of such a system can be made small. We have quantified this and have shown that in fact the trusted code can be made orders of magnitude smaller than in competing systems (JVMs). We have also analyzed the representation issues of the logical specification and shown how they relate to the size of the safety predicate and the proof checker. In our system the trusted code itself is based on a well understood and analyzed logical framework, which adds to our confidence of its correctness.

Bibliography

[1] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, July 2002.

[2] Andrew W. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE, 2001.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*. ACM Press, November 1999.

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, New York, January 2000. ACM Press.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, pages 657–683, September 2001.

[6] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A Trustworthy Proof Checker. Technical Report CS-TR-648-02, Princeton University, April 2002.

[7] Andrew W. Appel and Daniel C. Wang. JVM TCB: Measurements of the trusted computing base of Java virtual machines. Technical Report CS-TR-647-02, Princeton University, April 2002.

[8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998.

[9] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of USENIX Security*, August 2002.

[10] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, New York, June 2000. ACM Press.

[11] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press (New York, New York), October 1997.

[12] John DeTreville. Binder, a logic-based security language. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, page (to appear), May 2002.

[13] Edward W. Felten. Personal communication, April 2002.

[14] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[16] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher-order logic. In *17th International Conference on Automated Deduction*, pages 7–24, Berlin, June 2000. Springer-Verlag. LNAI 1831.

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, New York, January 1998. ACM Press.

[18] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[19] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, January 2001.

[20] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*, Berlin, July 1999. Springer-Verlag.

[21] Robert Pollack. How to believe a machine-checked proof. In Sambin and Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1996.

[22] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

Finding Counterexamples to Inductive Conjectures and Discovering Security Protocol Attacks

Graham Steel, Alan Bundy, Ewen Denney

Division of Informatics
University of Edinburgh
{grahams, bundy, ewd}@dai.ed.ac.uk

Abstract

We present an implementation of a method for finding counterexamples to universally quantified conjectures in first-order logic. Our method uses the proof by consistency strategy to guide a search for a counterexample and a standard first-order theorem prover to perform a concurrent check for inconsistency. We explain briefly the theory behind the method, describe our implementation, and evaluate results achieved on a variety of incorrect conjectures from various sources.

Some work in progress is also presented: we are applying the method to the verification of cryptographic security protocols. In this context, a counterexample to a security property can indicate an attack on the protocol, and our method extracts the trace of messages exchanged in order to effect this attack. This application demonstrates the advantages of the method, in that quite complex side conditions decide whether a particular sequence of messages is possible. Using a theorem prover provides a natural way of dealing with this. Some early results are presented and we discuss future work.

1 Introduction

Inductive theorem provers are frequently employed in the verification of programs, algorithms and protocols. However, programs and algorithms often contain bugs, and protocols may be flawed, causing the proof attempt to fail. It can be hard to interpret a failed proof attempt: it may be that some additional lemmas need to be proved or a generalisation made. In this situation, a tool which can not only detect an incorrect conjecture, but also supply a counterexample in order to allow the user to identify the bug or flaw, is potentially very valuable. The problem of cryptographic security protocol verification is a specific area in which incorrect conjectures are of great consequence. If a security conjecture turns out to be false, this can indicate an attack on the protocol. A counterexample can help the user to see how the protocol can be attacked. Incorrect conjectures also arise in automatic inductive theorem provers where generalisations are speculated by the system. Often we encounter the problem of over-generalisation: the speculated formula is not a theorem. A method for detecting these over-generalisations is required.

Proof by consistency is a technique for automating inductive proofs in first-order logic. Originally developed to prove correct theorems, this technique has the property of being refutation complete, i.e. it is able to refute in finite time conjectures which are inconsistent with the set of hypotheses. When originally proposed, this technique was of limited applicability. Recently, Comon and Nieuwenhuis have drawn together and extended previous research to show how it may be more generally applied, [10]. They describe an experimental implementation of the inductive completion part of the system. However, the check for refutation or consistency was not implemented. This check is necessary in order to ensure a theorem is correct, and to automatically refute an incorrect conjecture. We have implemented a novel system integrating Comon and Nieuwenhuis' experimental prover with a concurrent check for inconsistency. By carrying out the check in parallel, we are able to refute incorrect conjectures in cases where the inductive completion process fails to terminate. The parallel processes communicate via sockets using Linda, [8].

The ability of the technique to prove complex inductive theorems is as yet unproven. That does not concern us here – we are concerned to show that it provides an efficient and effective method for refuting *incorrect* conjectures. However, the ability to prove at least many small theorems helps alleviate a problem reported in Protzen's work on disproving conjectures, [22] – that the system terminates only at its depth limit in the case of a small unsatisfiable formula, leaving the user or proving system none the wiser.

We have some early results from our work in progress, which is to apply the technique to the aforementioned problem of cryptographic security protocol verification. These protocols often have subtle flaws in them that are not detected for years after they have been proposed. By devising a first-order version of Paulson's inductive for-

malism for the protocol verification problem, [21], and applying our refutation system, we can not only detect flaws but also automatically generate the sequence of messages needed to expose these flaws. By using an inductive model with arbitrary numbers of agents and runs rather than the finite models used in most model-checking methods, we have the potential to synthesise parallel session and replay attacks where a single principal may be required to play multiple roles in the exchange.

In the rest of the paper, we first review the literature related to the refutation of incorrect conjectures and proof by consistency, then we briefly examine the Comon-Nieuwenhuis method. We describe the operation of the system, relating it to the theory, and present and evaluate the results obtained so far. The system has been tested on a number of examples from various sources including Protzen's work [22], Reif et al.'s, [24], and some of our own. Our work in progress on the application of the system to the cryptographic protocol problem is then presented. Finally, we describe some possible further work and draw some conclusions.

2 Literature Review

2.1 Refuting Incorrect Conjectures

At the CADE-15 workshop on proof by mathematical induction, it was agreed that the community should address the issue of dealing with non-theorems as well as theorems[1]. However, relatively little work on the problem has since appeared. In the early nineties Protzen presented a sound and complete calculus for the refutation of faulty conjectures in theories with free constructors and complete recursive definitions, [22]. The search for the counterexample is guided by the recursive definitions of the function symbols in the conjecture. A depth limit ensures termination when no counterexample can be found.

More recently, Reif et al., [25], have implemented a method for counterexample construction that is integrated with the interactive theorem prover KIV, [23]. Their method incrementally instantiates a formula with constructor terms and evaluates the formulae produced using the simplifier rules made available to the system during proof attempts. A heuristic strategy guides the search through the resulting subgoals for one that can be reduced to $false$. If such a subgoal is not found, the search terminates when all variables of generated sorts have been instantiated to constructor terms. In this case the user is left with a model condition, which must be used to decide whether the instantiation found is a valid counterexample.

Ahrendt has proposed a refutation method using model construction techniques, [1]. This is restricted to free datatypes, and involves the construction of a set of suitable clauses to send to a model generation prover. As first reported, the approach was not able in general to find a refutation in finite time, but new work aims to address this problem, [2].

2.2 Proof by Consistency

Proof by consistency is a technique for automating inductive proof. It has also been called *inductionless induction*, and *implicit induction*, as the actual induction rule used is described implicitly inside a proof of the conjecture's consistency with the set of hypotheses. Recent versions of the technique have been shown to be *refutation complete*, i.e. are guaranteed to detect non-theorems in finite time.[2] The proof by consistency technique was developed to solve problems in equational theories, involving a set of equations defining the *initial model*[3], E. The first version of the technique was proposed by Musser, [20], for equational theories with a *completely defined equality predicate*, This requirement placed a strong restriction on the applicability of the method. The completion process used to deduce consistency was the Knuth-Bendix algorithm, [17].

Huet and Hullot [14] extended the method to theories with free constructors, and Jouannaud and Kounalis, [15], extended it further, requiring that E should be a convergent rewrite system. Bachmair, [4], proposed the first refutationally complete deduction system for the problem, using a *linear strategy* for inductive completion. This is a restriction of the Knuth-Bendix algorithm which entails only examining overlaps between axioms and conjectures. The key advantage of the restricted completion procedure was its ability to cope with unoriented equations. The refutational completeness of the procedure was a direct result of this.

The technique has been extended to the non-equational case. Ganzinger and Stuber, [12], proposed a method for proving consistency for a set of first-order clauses with equality using a refutation complete linear system. Kounalis and Rusinowitch, [18], proposed an extension to conditional theories, laying the foundations for the method implemented in the SPIKE theorem, [6]. Ideas from the proof by consistency technique have been used in other induction methods, such as cover set induction, [13], and test set induction, [5].

2.3 Cryptographic Security Protocols

Cryptographic protocols are used in distributed systems to allow agents to communicate securely. Assumed to be

[1] The minutes of the discussion are available from `http://www.cee.hw.ac.uk/~air/cade15/cade-15-mind-ws-session-3.html`.

[2] Such a technique must necessarily be incomplete with respect to proving theorems correct, by Gödel's incompleteness theorem.

[3] The initial or standard model is the minimal Herbrand model. This is unique in the case of a purely equational specification.

present in the system is a spy, who can see all the traffic in the network and may send malicious messages in order to try and impersonate users and gain access to secrets. Clark and Jacob's survey, [9], and Anderson and Needham's article, [3], are good introductions to the field.

Although security protocols are usually quite short, typically 2–5 messages, they often have subtle flaws in them that may not be discovered for many years. Researchers have applied various formal methods techniques to the problem, to try to find attacks on faulty protocols, and to prove correct protocols secure. These approaches include belief logics such as the so-called BAN logic, [7], state-machines, [11, 16], model-checking, [19], and inductive theorem proving, [21]. Each approach has its advantages and disadvantages. For example, the BAN logic is attractively simple, and has found some protocol flaws, but has missed others. The model checking approach can find flaws very quickly, but can only be applied to finite (and typically very small) instances of the protocol. This means that if no attack is found, there may still be an attack upon a larger instance. Modern state machine approaches can also find and exhibit attacks quickly, but require the user to choose and prove lemmas in order to reduce the problem to a tractable finite search space. The inductive method deals directly with the infinite state problem, and assumes an arbitrary number of protocol participants, but proofs are tricky and require days or weeks of expert effort. If a proof breaks down, there are no automated facilities for the detection of an attack.

3 The Comon-Nieuwenhuis Method

Comon and Nieuwenhuis, [10], have shown that the previous techniques for proof by consistency can be generalised to the production of a first-order axiomatisation $\mathcal{A}$ of the minimal Herbrand model such that $\mathcal{A} \cup E \cup C$ is consistent if and only if C is an inductive consequence of E. With $\mathcal{A}$ satisfying the properties they define as an *I-Axiomatisation*, inductive proofs can be reduced to first-order consistency problems and so can be solved by any saturation based theorem prover. We give a very brief summary of their results here. Suppose I is, in the case of Horn or equational theories, the unique minimal Herbrand model, or in the case of non-Horn theories, the so-called *perfect model* with respect to a total ordering on terms, $\succ$[4]:

Definition 1 *A set of first-order formulae $\mathcal{A}$ is an* I-Axiomatisation *of I if*

1. *$\mathcal{A}$ is a set of purely universally quantified formulae*
2. *I is the only Herbrand model of $E \cup \mathcal{A}$ up to isomorphism.*

[4]Saturation style theorem proving always requires that we have such an ordering available.

An I-Axiomatisation is normal *if $\mathcal{A} \models s \neq t$ for all pairs of distinct normal terms s and t*

The I-Axiomatisation approach produces a clean separation between the parts of the system concerned with inductive completion and inconsistency detection. Completion is carried out by a saturation based theorem prover, with inference steps restricted to those produced by conjecture superposition, a restriction of the standard superposition rule. Only overlaps between conjecture clauses and axioms are considered. Each non-redundant clause derived is checked for consistency against the I-Axiomatisation. If the theorem prover terminates with saturation, the set of formulae produced comprise a *fair induction derivation*. The key result of the theory is this:

Theorem 1 *Let $\mathcal{A}$ be a normal I-Axiomatisation, and $C_0, C_1, \ldots$ be a fair induction derivation. Then $I \models C_0$ iff $\mathcal{A} \cup \{c\}$ is consistent for all clauses c in $\bigcup_i C_i$.*

This theorem is proved in [10]. Comon and Nieuwenhuis have shown that this conception of proof by consistency generalises and extends earlier approaches. An equality predicate as defined by Musser, a set of free constructors as proposed by Huet and Hullot or a ground reducibility predicate as defined by Jouannaud and Kounalis could all be used to form a suitable I-Axiomatisation. The technique is also extended beyond ground convergent specifications (equivalent to saturated specifications for first-order clauses) as required in [15, 4, 12]. Previous methods, e.g. [6], have relaxed this condition by using conditional equations. However a ground convergent rewrite system was still required for deducing inconsistency. Using the I-Axiomatisation method, conjectures can be proved or refuted in (possibly non-free) constructor theories which cannot be specified by a convergent rewrite system.

Whether these extensions to the theory allow larger theorems to be *proved* remains to be seen, and is not of interest to us here. We are interested in how the wider applicability of the method can allow us to investigate the ability of the proof by consistency technique to root out a counterexample to realistic *incorrect* conjectures.

4 Implementation

Figure 1 illustrates the operation of our system. The input is an inductive problem in `Saturate` format and a normal I-Axiomatisation (see Definition 1, above). The version of `Saturate` customised by Nieuwenhuis for implicit induction (the right hand box in the diagram) gets the problem file only, and proceeds to pursue inductive completion, i.e. to derive a fair induction derivation. Every non-redundant clause generated is passed via the server to the refutation control program (the leftmost box). For every new clause received, this program generates a

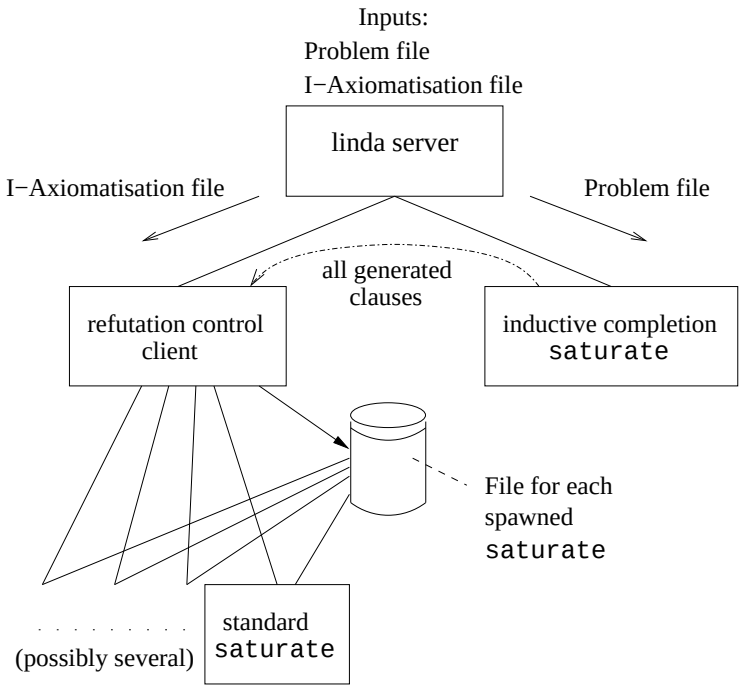


Figure 1: System operation

problem file containing the I-Axiomatisation and the new clause, and spawns a standard version of `Saturate` to check the consistency of the file. Crucially, these spawned `Saturate`s are not given the original axioms – only the I-Axioms are required, by Theorem 1. This means that almost all of the search for an inconsistency is done by the prover designed for inductive problems and the spawned `Saturate`s are just used to check for inconsistencies between the new clauses and the I-Axiomatisation. This should lead to a false conjecture being refuted after fewer inference steps have been attempted than if the conjecture had been given to a standard first-order prover together with all the axioms and I-Axioms. We evaluate this in the next section.

If, at any time, a refutation is found by a spawned prover, the proof is written to a file and the completion process and all the other spawned `Saturate` processes are killed. If completion is reached by the induction prover, this is communicated to the refutation control program, which will then wait for the results from the spawned processes. If they all terminate with saturation, then there are no inconsistencies, and so the theorem has been proved (by Theorem 1).

There are several advantages to the parallel architecture we have employed. One is that it allows us to refute incorrect conjectures even if the inductive completion process does not terminate. This would also be possible by modifying the main induction `Saturate` to check each clause in tern, but this would result in a rather messy and unwieldy program. Another advantage is that we are able to easily devote a machine solely to inductive completion in the case of harder problems. It is also very convenient when testing a new model to be able to just look at the deduction process before adding the consistency check later on, and we preserve the attractive separation in the theory between the deduction and the consistency checking processes.

A disadvantage of our implementation is that launching a new `Saturate` process to check each clause against the I-Axiomatisation generates some overheads in terms of disk access etc. In our next implementation, when a spawned prover reaches saturation (i.e. no inconsistencies), it will clear its database and ask the refutation control client for another clause to check, using the existing sockets mechanism. This will cut down the amount of memory and disk access required. A further way to reduce the consistency checking burden is to take advantage of knowledge about the structure of the I-Axiomatisation for simple cases. For example, in the case of a free constructor specification, the I-Axiomatisation will consist of clauses specifying the inequality of non-identical constructor terms. Since it will include no rules referring to defined symbols, it is sufficient to limit the consistency check to generated clauses containing only constructors and variables.

Table 1: Sample of results. *In the third column, the first number shows the number of clauses derived by the inductive completion prover, and the number in brackets indicates the number of clauses derived by the parallel checker to spot the inconsistency. The fourth column shows the number of clauses derived by an unmodified first-order prover when given the conjecture, axioms and I-Axioms all together.*

Problem	Counterexample found	No. of clauses derived to find refutation	No. of clauses derived by a standard prover
$\forall N, M.\neg(s(N)+M=s(0))$	$N=0, M=0$	2(+0)	2
$X \neq Y \wedge X \neq 0 \wedge Y \neq 0$ $\Rightarrow (X \geq Y \wedge Y \neq 0) \vee$ $(X \neq 0 \wedge Y = 0)$	$X = s(0)$, $Y = s(s(X))$	4 (+3)	6
$app(K, L) = app(L, K)$	$K = 0, L = s(X)$	9(+11)	stuck in loop
$sort(l_1) \wedge l_2 = ap(l_1, [head(l_3)])$ $\wedge length(l_3) \geq 2 * length(l_1)$ $\wedge l_3 \neq nil \wedge$ $member(head(l_1), tail(l_3))$ $\Rightarrow sort(l_2)$	$l_1 = [s(X)]$, $l_2 = [s(X), 0]$ $l_3 = [0, s(X)\|Y]$	55(+1)	76
All graphs are acyclic	$[e(a, a)]$	99	123
All loopless graphs are acyclic	$[e(s(a), a), e(a, s(a))]$	178	2577
$gcd(X, X) = 0$	$X = s(0)$	17(+2)	29
Impossibility property for Neuman-Stubblefield key exchange protocol	$[msg(1), msg(2),$ $msg(3), msg(4)]$	866 (+0)	1733
Authenticity property for simple protocol from [9]	see section 6	730(+1)	3148

5 Evaluation of Results

Table 1 shows a sample of results achieved so far. The first three examples are from Protzen's work, [22], the next two from Reif et al.'s, [25], and the last three are from our work. The gcd example is included because previous methods of proof by consistency could not refute this conjecture. Comon and Nieuwenhuis showed how it could be tackled, [10], and here we confirm that their method works. The last two conjectures are about properties of security protocols. The 'impossibility property' states that no trace reaches the end of a protocol. Its refutation comprises the proof of a possibility property, which is the first thing proved about a newly modelled protocol in Paulson's method, [21]. The last result is the refutation of an authenticity property, indicating an attack on the protocol. This protocol is a simple example included in Clark's survey, [9], for didactic purposes, but requires that one principal play both roles in a protocol run. More details are given in section 6.

Our results on Reif et al.'s examples do not require the user to verify a model condition, as the system described in their work does. Interestingly, the formula remaining as a model condition in their runs is often the same as the formula which gives rise to the inconsistency when checked against the I-Axiomatisation in our runs. This is because the KIV system stops when it derives a term containing just constructors and variables. In such a case, our I-Axiomatisation would consist of formulae designed to check validity of these terms. This suggests a way to automate the model condition check in the KIV system.

On comparing the number of clauses derived by our system and the number of clauses required by a standard first-order prover (SPASS), we can see that the proof by consistency strategy does indeed cut down on the number of inferences required. This is more evident in the larger examples. Also, the linear strategy allows us to cope with commutativity conjectures, like the third example, which cause a standard prover to go into a loop. We might ask: what elements of the proof by consistency technique are allowing us to make this saving in required inferences? One is the refutation completeness result for the linear strategy, so we know we need only consider overlaps between conjectures and axioms. Additionally, separating the I-Axioms from the theory axioms reduces the number of overlaps between conjectures and axioms to be considered each time. We also use the results about inductively complete positions for theories with free constructors, [10]. This applies to all the examples except those in graph theory, where we used Reif's formalism and hence did not have free constructors. This is the probable reason why, on these two examples, our system did not make as large a saving in

derived clauses.

The restriction to overlaps between conjectures and axioms is similar in nature to the so-called set of support strategy, using the conjecture as the initial supporting set. The restriction in our method is tighter, since we don't consider overlaps between formulae in the set of support. Using the set of support strategy with the standard prover on the examples in Table 1, refutations are found after deriving fewer clause than required by the standard strategy. However, performance is still not as good as for our system, particularly in the free constructor cases. The set of support also doesn't fix the problem of divergence on unoriented conjectures, like the commutativity example.

The efficiency of the method in terms of clauses derived compared to a standard prover looks good. However, actual time taken by our system is much longer than that for the standard SPASS. This is because the `Saturate` prover is rather old, and was not designed to be a serious tool for large scale proving. In particular, it does not utilise any term indexing techniques, and so redundancy checks are extremely slow. As an example, the impossibility property took about 50 minutes to refute in `Saturate`, but about 40 seconds in SPASS, even though more than twice as many clauses had to be derived. We used `Saturate` in our first system as Nieuwenhuis had already implemented the proof by consistency strategy in the prover. A reimplementation of the whole system using SPASS should give us even faster refutations, and is one of our next tasks.

Finally, we also tested the system on a number of small inductive theorems. Being able to prove small theorems allows us to attack a problem highlighted in Protzen's work: that if an candidate generalisation (say) is given to the counterexample finder and it returns a result saying that the depth limit was reached before a counterexample was found, the system is none the wiser as to whether the generalisation is worth pursuing. If we are able to prove at least small examples to be theorems, this will help alleviate the problem. Our results were generally good: 7 out of 8 examples we tried were proved, but one was missed. Comon intends to investigate the ability of the technique to prove more and larger theorems in future.

More details of the results including some sample runs and details of the small theorems proved can be found at `http://www.dai.ed.ac.uk/~grahams/linda`.

6 Application to Cryptographic Security Protocols

We now describe some work in progress on applying our technique to the cryptographic security protocol problem. As we saw in section 2.3, one of the main thrusts of research has been to apply formal methods to the problem. Researchers have applied techniques from model checking, theorem proving and modal logics amongst others. Much attention is paid to the modelling of the abilities of the spy in these models. However, an additional consideration is the abilities of the participants. Techniques assuming a finite model, with typically two agents playing distinct roles, often rule out the possibility of discovering a certain kind of parallel session attack, in which one participant plays both roles in the protocol. The use of an inductive model allows us to discover these kind of attacks. An inductive model also allows us to consider protocols with more than two participants, e.g. conference-key protocols.

Paulson's inductive approach has used been used to verify properties of several protocols, [21]. Protocols are formalised in typed higher-order logic as the set of all possible traces, a trace being a list of events like 'A sends message X to B'. This formalism is mechanised in the Isabelle/HOL interactive theorem prover. Properties of the security protocol can be proved by induction on traces. The model assumes an arbitrary number of agents, and any agent may take part in any number of concurrent protocol runs playing any role. Using this method, Paulson discovered a flaw in the simplified Otway-Rees shared key protocol, [7], giving rise to a parallel session attack where a single participant plays both protocol roles. However, as Paulson observed, a failed proof state can be difficult to interpret in these circumstances. Even an expert user will be unsure as to whether it is the proof attempt or the conjecture which is at fault. By applying our counterexample finder to these problems, we can automatically detect and present attacks when they exist.

Paulson's formalism is in higher-order logic. However, no 'fundamentally' higher-order concepts are used – in particular there is no unification of higher-order objects. Objects have types, and sets and lists are used. All this can be modelled in first-order logic. The security protocol problem has been modelled in first-order logic before, e.g. by Weidenbach, [26]. This model assumed a two agent model with just one available nonce[5] and key, and so could not detect the kind of parallel session attacks described. Our model allows an arbitrary number of agents to participate, playing either role, and using an arbitrary number of fresh nonces and keys.

6.1 Our Protocol Model

Our models aims to be as close as possible to a first-order version of Paulson's formalism. As in Paulson's model, agents, nonces and messages are free data types. This allows us to define a two-valued function eq which will tell us whether two pure constructor terms are equal or not. Since the rules defining eq are exhaustive, they also have the effect of suggesting instantiations where certain conditions must be met, e.g. if we require the identities of two agents to be distinct. The model is kept Horn by defining two-valued functions for checking the side conditions

[5] A nonce is a unique identifying number.

for a message to be sent, e.g. we define conditions for $member(X, L) = true$ and $member(X, L) = false$ using our eq function. This cuts down the branching rate of the search.

The intruder's knowledge is specified in terms of sets. Given a trace of messages exchanged, XT, we define $analz(XT)$ to be the least set including XT closed under projection and decryption by known keys. This is accomplished by using exactly the same rules as the Paulson model, [21, p. 12]. Then, we can define the messages the intruder may send, given a trace XT, as being members of the set $synth(analz(XT))$, where $synth(X)$ is the least set including agent names closed under pairing and encryption by known keys. Again this set is defined in our model with the same axioms that Paulson uses.

A trace of messages is modelled as a list. For a specific protocol, we generally require one axiom for each protocol message. These axioms take the form of rules with the informal interpretation, 'if XT is a trace containing message n addressed to agent xa, then the trace may be extended by xa responding with message $n + 1$'. Once again, this is very similar to the Paulson model.

An example illustrates some of these ideas. In Figure 2 we demonstrate the formalism of a very simple protocol included in Clark and Jacob's survey to demonstrate parallel session attacks, [9]. Although simple, the attack on the protocol does require principal A to play the role of both initiator and responder. It assumes that A and B already share a secure key, K_{AB}. N_A denotes a nonce generated by A.

In a symmetric key protocol, principals should respond to $key(A, B)$ and $key(B, A)$, as they are in reality the same. At the moment we model this with two possible rules for message 2, but it should be straightforward to extend the model to give a cleaner treatment of symmetric keys as sets of agents. Notice we allow a principal to respond many times to the same message, as Paulson's formalism does.

The second box, Figure 3, shows how the refutation of a conjectured security property leads to the discovery of the known attack. At the moment, choosing which conjectures to attempt to prove is tricky. A little thought is required in order to ensure that only a genuine attack can refute the conjecture. More details of our model for the problem, including the specification of intruder knowledge, can be found at `http://www.dai.ed.ac.uk/~grahams/linda`.

This application highlights a strength of our refutation system: in order to produce a backwards style proof, as Paulson's system does, we must apply rules with side conditions referring as yet uninstantiated variables. For example, a rule might be applied with the informal interpretation, 'if the spy can extract X from the trace of messages sent up to this point, then he can break the security conjecture'. At the time the rule is applied, X will be uninstantiated. Further rules instantiate parts of the trace, and side conditions are either satisfied and eliminated, or found to be unsatisfiable, causing the clauses containing the condition to be pruned off as redundant. The side conditions influence the path taken through the search space, as smaller formulae are preferred by the default heuristic in the prover. This means that some traces a naïve counterexample search might find are not so attractive to our system, e.g. a trace which starts with several principals sending message 1 to other principals. This will not be pursued at first, as all the unsatisfied side conditions will make this formula larger than others.

7 Further Work

Our first priority is to re-implement the system using SPASS, and then to carry out further experiments with larger false conjectures and more complex security protocols. This will allow us to evaluate the technique more thoroughly. A first goal is to rediscover the parallel session attack discovered by Paulson. The system should also be able to discover more standard attacks, and the Clark survey, [9], provides a good set of examples for testing. We will then try the system on other protocols and look for some new attacks. A key advantage of our security model is that it allows attacks involving arbitrary numbers of participants. This should allow us to investigate the security of protocols involving many participants in a single run, e.g. conference key protocols.

In future, we also intend to implement more sophisticated heuristics to improve the search performance, utilising domain knowledge about the security protocol problem. Heuristics could include eager checks for unsatisfiable side conditions. Formulae containing these conditions could be discarded as redundant. Another idea is to vary the weight ascribed to variables and function symbols, so as to make the system inclined to check formulae with predominantly ground variables before trying ones with many uninstantiated variables. This should make paths to attacks more attractive to the search mechanism, but some careful experimentation is required to confirm this.

The Comon-Nieuwenhuis technique has some remaining restrictions on applicability, in particular the need for *reductive definitions*, a more relaxed notion of reducibility than is required for ground convergent rewrite systems. It is quite a natural requirement that recursive function definitions should be reducing in some sense. For example, the model of the security protocol problem is reductive in the sense required by Comon and Nieuwenhuis. Even so, it should be possible to extend the technique for non-theorem detection in the case of non-reductive definitions, at the price of losing any reasonable chance of proving a theorem, but maintaining the search guidance given by the proof by consistency technique. This would involve allowing inferences by standard superposition if conjecture

The Clark-Jacob protocol demonstrating parallel session attacks. At the end of a run, A should now be assured of B's presence, and has accepted nonce N_A to identify authenticated messages.

1. $A \rightarrow B : \{\!|\ N_A |\!\}_{K_{AB}}$
2. $B \rightarrow A : \{\!|\ N_A + 1 |\!\}_{K_{AB}}$

Formula for modelling message 1 of the protocol. Informally: if XT is a trace, XA and XB agents, and XNA a number not appearing as a nonce in a previous run, then the trace may be extended by XA initiating a run, sending message 1 of the protocol to XB.

$$m(XT) = true \wedge agent(XA) = true \wedge agent(XB) = true \wedge number(XNA) = true$$
$$\wedge member(sent(X, Y, encr(nonce(XNA), K)), XT) = false \Rightarrow$$
$$m([sent(XA, XB, encr(nonce(XNA), key(XA, XB)))|XT]) = true$$

Formulae for message 2. Two formulae are used to make the response to the shared key symmetric (see text). Informally: if XT is a trace containing message 1 of the protocol addressed to agent XB, encrypted under a key he shares with agent XA, then the trace may be extended by agent XB responding with message 2.

$$member(sent(X, XB, encr(nonce(XNA), key(XA, XB))), XT) = true \wedge m(XT) = true \Rightarrow$$
$$m([sent(XB, XA, encr(s(nonce(XNA)), key(XA, XB)))|XT]) = true.$$

$$member(sent(X, XB, encr(nonce(XNA), key(XB, XA))), XT) = true \wedge m(XT) = true \Rightarrow$$
$$m([sent(XB, XA, encr(s(nonce(XNA)), key(XB, XA)))|XT]) = true.$$

Figure 2: The modelling of the Clark-Jacob protocol

The parallel session attack suggested by Clark and Jacob [9]. At the end of the attack, A believes B is operational. B may be absent or may no longer exist:

1. $A \rightarrow C_B : \{\!|\ N_A |\!\}_{K_{AB}}$
2. $C_B \rightarrow A : \{\!|\ N_A |\!\}_{K_{AB}}$
3. $A \rightarrow C_B : \{\!|\ s(N_A) |\!\}_{K_{AB}}$
4. $C_B \rightarrow A : \{\!|\ s(N_A) |\!\}_{K_{AB}}$

Below is the incorrect security conjecture, the refutation of which gives rise to the attack above. Informally this says, 'for all valid traces T, if A starts a run with B using nonce N_A, and receives the reply $s(N_A)$ from principal X, and no other principal has sent a reply, then the reply must have come from agent B.'

$$member(sent(XA, XB, encr(nonce(XNA), K), XT) = true$$
$$\wedge XT = [sent(X, XA, encr(s(nonce(XNA)), K)|T]$$
$$\wedge member(sent(Y, XA, encr(s(nonce(XNA)), K), T) = false$$
$$\wedge m(XT) = true \Rightarrow eq(X, XB) = true$$

The final line of output from the system, giving the attack.

```
c(sent(spy,a,encr(s(nonce(0)),key(a,s(a)))),
c(sent(a,s(a),encr(s(nonce(0)),key(a,s(a)))),
c(sent(spy,a,encr(nonce(0),key(a,s(a)))),
c(sent(a,s(a),encr(nonce(0),key(a,s(a)))),nil))))
```

Figure 3: The attack and its discovery

superposition is not applicable.

8 Conclusions

In this paper we have presented a working implementation of a novel method for investigating an inductive conjecture, with a view to proving it correct or refuting it as false. We are primarily concerned with the ability of the system to refute false conjectures, and have shown results from testing on a variety of examples. These have shown that our parallel inductive completion and consistency checking system requires considerably fewer clauses to be derived than a standard first-order prover does when tackling the whole problem at once. The application of the technique to producing attacks on faulty cryptographic security protocols looks promising, and the system has already synthesised an attack of a type many finite security models will not detect. We intend to produce a faster implementation using the SPASS theorem prover, and then to pursue this application further.

Bibliography

[1] W. Ahrendt. A basis for model computation in free data types. In *CADE-17, Workshop on Model Computation - Principles, Algorithmns, Applications*, 2000.

[2] W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In *CADE-18*, 2002.

[3] R. Anderson and R. Needham. *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, chapter Programming Satan's Computer, pages 426–440. Springer, 1995.

[4] L. Bachmair. *Canonical Equational Proofs*. Birkhauser, 1991.

[5] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. Rapport de Recherche 1663, INRIA, April 1992.

[6] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[7] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[8] N. Carreiro and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[9] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[10] H. Comon and R. Nieuwenhuis. Induction = I-Axiomatization + First-Order Consistency. *Information and Computation*, 159(1-2):151–186, May/June 2000.

[11] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[12] H. Ganzinger and J. Stuber. *Informatik — Festschrift zum 60. Geburtstag von Günter Hotz*, chapter Inductive theorem proving by consistency for first-order clauses, pages 441–462. Teubner Verlag, 1992.

[13] M. S. Krishnamoorthy H. Zhang, D. Kapur. A mechanizable induction principle for equational specifications. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.

[14] G. Huet and J. Hullot. Proofs by induction in equational theories with constructors. *Journal of the Association for Computing Machinery*, 25(2), 1982.

[15] J.-P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. *Information and Computation*, 82(1), 1989.

[16] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7:79–130, 1994.

[17] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[18] E. Kounalis and M. Rusinowitch. A mechanization of inductive reasoning. In AAAI Press and MIT Press, editors, *Proceedings of the American Association for Artificial Intelligence Conference, Boston*, pages 240–245, July 1990.

[19] G. Lowe. Breaking and fixing the Needham Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.

[20] D. Musser. On proving inductive properties of abstract data types. In *Proceedings 7th ACM Symp. on Principles of Programming Languages*, pages 154–162. ACM, 1980.

[21] L.C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[22] M. Protzen. Disproving conjectures. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 340–354, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[23] W. Reif. The KIV Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009. Springer Verlag, 1995.

[24] W. Reif, G. Schellhorn, and A. Thums. Fehlersuche in formalen Spezifikationen. Technical Report 2000-06, Fakultät fur Informatik, Universität Ulm, Germany, May 2000. (In German).

[25] W. Reif, G. Schellhorn, and A. Thums. Flaw detection in formal specifications. In *IJCAR'01*, pages 642–657, 2001.

[26] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 314–328, Trento, Italy, July 1999. Springer-Verlag.

Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning*

Alessandro Armando Luca Compagna

DIST — Università degli Studi di Genova
Viale Causa 13 – 16145 Genova, Italy
{armando,compa}@dist.unige.it

Abstract

We provide a fully automatic translation from security protocol specifications into propositional logic which can be effectively used to find attacks to protocols. Our approach results from the combination of a reduction of protocol insecurity problems to planning problems and well-known SAT-reduction techniques developed for planning. We also propose and discuss a set of transformations on protocol insecurity problems whose application has a dramatic effect on the size of the propositional encoding obtained with our SAT-compilation technique. We describe a model-checker for security protocols based on our ideas and show that attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

Keywords: Foundation of verification; Confidentiality and authentication; Intrusion detection.

1 Introduction

Even under the assumption of perfect cryptography, the design of security protocols is notoriously error-prone. As a consequence, a variety of different protocol analysis techniques has been put forward [3, 4, 8, 10, 12, 16, 19, 22, 23]. In this paper we address the problem of translating protocol insecurity problems into propositional logic in a fully automatic way with the ultimate goal to build an automatic model-checker for security protocols based on state-of-the-art SAT solvers. Our approach combines a reduction of protocol insecurity problems to planning problems[1] with well-known SAT-reduction techniques developed for planning. At the core of our technique is a set of transformations whose application to the input protocol insecurity problem has a dramatic effect on the size of the propositional formulae obtained. We present a model-checker for security protocols based on our ideas and show that—using our tool—attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

2 Security Protocols and Protocol Insecurity Problems

In this paper we concentrate our attention on error detection of authentication protocols (see [7] for a survey). As a simple example consider the following one-way authentication protocol:

$$(1)\ \ A \rightarrow B : \{N_a\}_{K_{ab}}$$
$$(2)\ \ B \rightarrow A : \{f(N_a)\}_{K_{ab}}$$

where N_a is a nonce[2] generated by Alice, K_{ab} is a symmetric key, f is a function known to Alice and Bob, and $\{x\}_k$ denotes the result of encrypting text x with key k. Successful execution of the protocol should convince Alice that she has been talking with Bob, since only Bob could have formed the appropriate response to the message issued in (1). In fact, Ivory can deceit Alice into believing that she is talking with Bob whereas she is talking with her. This is achieved by executing concurrently two sessions of the protocol and using messages from one session to form messages in the other as illustrated by the following protocol trace:

$$(1.1)\ \ A \rightarrow I(B) : \{N_a\}_{K_{ab}}$$
$$(2.1)\ \ I(B) \rightarrow A : \{N_a\}_{K_{ab}}$$
$$(2.2)\ \ A \rightarrow I(B) : \{f(N_a)\}_{K_{ab}}$$
$$(1.2)\ \ I(B) \rightarrow A : \{f(N_a)\}_{K_{ab}}$$

Alice starts the protocol with message (1.1). Ivory intercepts the message and (pretending to be Bob) starts a second session with Alice by replaying the received message—cf. step (2.1). Alice replies to this message

*This work has been supported by the Information Society Technologies Programme, FET Open Assessment Project "AVISS" (Automated Verification of Infinite State Systems), IST-2000-26410.

[1] The idea of regarding security protocol analysis as a planning problem is not new. To our knowledge it is also been proposed in [1].

[2] *Nonces* are numbers generated by principals that are intended to be used *only once*.

with message (2.2). But this is exactly the message Alice is waiting to receive in the first protocol session. This allows Ivory to finish the first session by using it—cf. (1.2). At the end of the above steps Alice believes she has been talking with Bob, but this is obviously not the case.

A problem with the above rule-based notation to specify security protocols is that it leaves implicit many important details such as the shared information and how the principals should react to messages of an unexpected form. This kind of description is therefore usually supplemented with explanations in natural language which in our case explain that N_a is a nonce generated by Alice, that f is a function known to the honest participants, and that K_{ab} is a shared key.

To cope with the above difficulties and pave the way to the formal analysis of security protocols a set of models and specification formalisms as well as translators from high-level languages (similar to the one we used above to introduce our example) into these formalisms have been put forward. For instance, Casper [18] compiles high-level specifications into CSP, whereas CAPSL [5] and the AVISS tool [2] compile high-level specifications into formalisms based on multiset rewriting inspired by [6].

2.1 The Model

We model the concurrent execution of a protocol by means of a state transition system. Following [16], we represent states by sets of atomic formulae called *facts* and transitions by means of rewrite rules over sets of facts. For the simple protocol above, facts are built out of a first-order sorted signature with sorts `user`, `number`, `key`, `func`, `text` (super-sort of all the previous sorts), `int`, `session`, `nonceid`, and `list_of text`. The constants 0, 1, and 2 (of sort `int`) denote protocols steps, $\underline{1}$ and $\underline{2}$ (of sort `session`) denote session instances, a and b (of sort `user`) denote honest participants, k_{ab} (of sort `key`) denotes a symmetric key and na (of sort `nonceid`) is a nonce identifier. The function symbol $\{\_\}_\_ : \texttt{text} \times \texttt{key} \to \texttt{text}$ denotes the encryption function, $f : \texttt{number} \to \texttt{func}$ denotes the function known to the honest participants, $nc : \texttt{nonceid} \times \texttt{session} \to \texttt{number}$, and $s : \texttt{session} \to \texttt{session}$ are nonce and session constructors respectively. The predicate symbols are i of arity `text`, fresh of arity `number`, m of arity $\texttt{int} \times \texttt{user} \times \texttt{user} \times \texttt{text}$, and w of arity $\texttt{int} \times \texttt{user} \times \texttt{user} \times \texttt{list\_of text} \times \texttt{list\_of text} \times \texttt{session}$:

- $i(t)$ means that the intruder knows t.
- $\mathit{fresh}(n)$ means that n has not been used yet.
- $m(j, s, r, t)$ means that principal s has (supposedly)[3] sent message t to principal r at protocol step j.
- $w(j, s, r, ak, ik, c)$ represents the state of execution of principal r at step j of session c; in particular it means that r knows the terms stored in the lists ak (*acquired knowledge*) and ik (*initial knowledge*) at step j of session c, and—if $j \neq 0$—also that a message from s to r is awaited for step j to be executed.

Initial States. The initial state of the system is:[4]

$$w(0, a, a, [\,], [a, b, k_{ab}], \underline{1}) \,.\, w(1, a, b, [\,], [b, a, k_{ab}], \underline{1}) \tag{1}$$

$$.\, w(0, b, b, [\,], [b, a, k_{ab}], \underline{2}) \,.\, w(1, b, a, [\,], [a, b, k_{ab}], \underline{2}) \tag{2}$$

$$.\, \mathit{fresh}(nc(na, \underline{1})) \,.\, \mathit{fresh}(nc(na, s(\underline{1}))) \tag{3}$$

$$.\, \mathit{fresh}(nc(na, \underline{2})) \,.\, \mathit{fresh}(nc(na, s(\underline{2}))) \tag{4}$$

$$.\, i(a) \,.\, i(b) \tag{5}$$

Facts (1) represent the initial state of principals a and b (as initiator and responder, resp.) in session $\underline{1}$. Dually, facts (2) represent the initial state of principals b and a (as responder and initiator, resp.) in session $\underline{2}$. Facts (3) and (4) state the initial freshness of the nonces. Facts (5) represent the information initially known by the intruder.

Rewrite rules over sets of facts are used to specify the transition system evolves.

Protocol Rules. The following rewrite rule models the activity of sending the first message:

$$\begin{aligned} &w(0, A, A, [\,], [A, B, K_{ab}], C) \,.\, \mathit{fresh}(nc(na, C)) \\ &\quad \xrightarrow{step_1(A,B,C,K_{ab})} m(1, A, B, \{nc(na, C)\}_{K_{ab}}) \\ &\qquad .\, w(2, B, A, [nc(na, C)], [A, B, K_{ab}], C) \end{aligned} \tag{6}$$

Notice that nonce $nc(na, C)$ is added to the acquired knowledge of A for subsequent use. The receipt of the message and the reply of the responder is modeled by:

$$\begin{aligned} &m(1, A, B, \{nc(ID, C1)\}_{K_{ab}}) \\ &.\, w(1, A, B, [\,], [B, A, K_{ab}], C) \xrightarrow{step_2(A,B,C,C1,K_{ab},ID)} \\ &\qquad m(2, B, A, \{f(nc(ID, C1))\}_{K_{ab}}) \\ &\qquad\quad .\, w(1, A, B, [\,], [B, A, K_{ab}], s(C)) \end{aligned} \tag{7}$$

The final step of the protocol is modeled by:

$$\begin{aligned} &m(2, B, A, \{f(nc(ID, C1))\}_{K_{ab}}) \\ &\quad .\, w(2, B, A, [nc(ID, C1)], [A, B, K_{ab}], C) \\ &\qquad\qquad \xrightarrow{step_3(A,B,C,C1,K_{ab},ID)} \\ &\qquad\qquad .\, w(0, A, A, [\,], [A, B, K_{ab}], s(C)) \end{aligned} \tag{8}$$

[3] As we will see, since the intruder may fake other principals' identity, the message might have been sent by the intruder.

[4] To improve readability we use the "$.$" operator as set constructor. For instance, we write "$x \,.\, y \,.\, z$" to denote the set $\{x, y, z\}$.

Intruder Rules. There are also rules specifying the behavior of the intruder. In particular the intruder is based on the model of Dolev and Yao [11]. For instance, the following rule models the ability of the intruder of diverting the information exchanged by the honest participants:

$$m(J,S,R,T) \xrightarrow{divert(J,R,S,T)} i(S) \,.\, i(R) \,.\, i(T) \quad (9)$$

The ability of encrypting and decrypting messages is modeled by:

$$i(T) \,.\, i(K) \xrightarrow{encrypt(K,T)} i(T) \,.\, i(K) \,.\, i(\{T\}_K) \quad (10)$$

$$i(\{T\}_K) \,.\, i(K) \xrightarrow{decrypt(K,T)} i(K) \,.\, i(T) \quad (11)$$

Finally, the intruder can send arbitrary messages possibly faking somebody else's identity in doing so:

$$i(T) \,.\, i(S) \,.\, i(R) \xrightarrow{fake_1(R,S,T)} i(T) \,.\, i(S) \,.\, i(R) \,.\, m(1,S,R,T) \quad (12)$$

$$i(T) \,.\, i(S) \,.\, i(R) \xrightarrow{fake_2(R,S,T)} i(T) \,.\, i(S) \,.\, i(R) \,.\, m(2,S,R,T) \quad (13)$$

Bad States. A security protocol is intended to enjoy a a specific security property. In our example this property is the ability of authenticating Bob to Alice. A security property can be specified by providing a set of "bad" states, i.e. states whose reachability implies a violation of the property. For instance, it is easy to see that any state containing both $w(0,a,a,[\,],[a,b,k_{ab}],s(\underline{1}))$ (i.e. Alice has finished the first run of session $\underline{1}$) and $w(1,a,b,[\,],[b,a,k_{ab}],\underline{1})$ (i.e. Bob is still at the beginning of session $\underline{1}$) witnesses a violation of the expected authentication property of our simple protocol and therefore it should be considered as a bad state.

2.2 Protocol Insecurity Problems

The above concepts can be recast into the concept of protocol insecurity problem. A *protocol insecurity problem* is a tuple $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$ where $\mathcal{S}$ is a set of atomic formulae of a sorted first-order language called *facts*, $\mathcal{L}$ is a set of function symbols called *rule labels*, and $\mathcal{R}$ is a set of rewrite rules of the form $L \xrightarrow{\ell} R$, where L and R are finite subsets of $\mathcal{S}$ such that the variables occurring in R occur also in L, and ℓ is an expression of the form $l(\vec{x})$ where $l \in \mathcal{L}$ and $\vec{x}$ is the vector of variables obtained by ordering lexicographically the variables occurring in L. Let S be a state and $(L \xrightarrow{\ell} R) \in \mathcal{R}$, if σ is a substitution such that $L\sigma \subseteq S$, then one possible next state of S is $S' = (S \setminus L\sigma) \cup R\sigma$ and we indicate this with $S \xrightarrow{\ell\sigma} S'$. We assume the rewrite rules are *deterministic* i.e. if $S \xrightarrow{\ell\sigma} S'$ and $S \xrightarrow{\ell\sigma} S''$, then $S' \equiv S''$. The components $\mathcal{I}$ and $\mathcal{B}$ of a protocol insecurity problem are the initial state and a sets of states whose elements represent the bad states of the protocol respectively. A *solution to a protocol insecurity problem* Ξ (i.e. an attack to the protocol) is a sequence of states $S_1, \ldots, S_n$ such that $S_i \xrightarrow{\ell_i\sigma_i} S_{i+1}$ for $i = 1, \ldots, n$ and $\mathcal{I} \equiv S_1$, and there exists $S_{\mathcal{B}} \in \mathcal{B}$ such that $S_{\mathcal{B}} \subseteq S_n$.

3 Automatic SAT-Compilation of Protocol Insecurity Problems

Our proposed reduction of protocol insecurity problems to propositional logic is carried out in two steps. Protocol insecurity problems are first translated into planning problems which are in turn encoded into propositional formulae.

A *planning problem* is a tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$, where $\mathcal{F}$ and $\mathcal{A}$ are disjoint sets of variable-free atomic formulae of a sorted first-order language called *fluents* and *actions* respectively; Ops is a set of expressions of the form

$$op(Act, Pre, Add, Del)$$

where $Act \in \mathcal{A}$ and Pre, Add, and Del are finite sets of fluents such that $Add \cap Del = \emptyset$; I and G are boolean combinations of fluents representing the initial state and the final states respectively. A state is represented by a set of fluents. An action is applicable in a state S iff the action preconditions occur in S and the application of the action leads to a new state obtained from S by removing the fluents in Del and adding those in Add. A *solution to a planning problem* Π is a sequence of actions whose execution leads from the initial state to a final state and the precondition of each action appears in the state to which it applies.

3.1 Encoding Planning Problems into SAT

Let $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G \rangle$ be a planning problem with finite $\mathcal{F}$ and $\mathcal{A}$ and let n be a positive integer, then it is possible to build a set of propositional formulae Φ^n_Π such that any model of Φ^n_Π corresponds to a partial-order plan of length n which can be linearized into a solution of Π. The encoding of a planning problem into a set of SAT formulae can be done in a variety of ways (see [17, 13] for a survey). The basic idea is to add an additional time-index to the actions and fluents to indicate the state at which the action begins or the fluent holds. Fluents are thus indexed by 0 through n and actions by 0 through $n-1$. If p is a fluent or an action and i is an index in the appropriate range, then $i{:}p$ is the corresponding time-indexed propositional variable.

The set of formulae Φ^n_Π is the smallest set (intended conjunctively) such that:

- **Initial State Axioms:** $0{:}I \in \Phi^n_\Pi$;
- **Goal State Axioms:** $n{:}G \in \Phi^n_\Pi$;
- **Universal Axioms:** for each $op(\alpha, Pre, Add, Del) \in Ops$ and $i = 0, \ldots, n-1$:

$$(i{:}\alpha \supset \bigwedge\{i{:}p \mid p \in Pre\}) \in \Phi^n_\Pi$$
$$(i{:}\alpha \supset \bigwedge\{(i+1){:}p \mid p \in Add\}) \in \Phi^n_\Pi$$
$$(i{:}\alpha \supset \bigwedge\{\neg(i+1){:}p \mid p \in Del\}) \in \Phi^n_\Pi$$

- **Explanatory Frame Axioms:** for all fluents f and $i = 0, \ldots, n-1$:

$$(i{:}f \wedge \neg(i+1){:}f) \supset \bigvee \{i{:}\alpha \mid op(\alpha, Pre, Add, Del) \in Ops, f \in Del\} \in \Phi^n_\Pi$$

$$(\neg i{:}f \wedge (i+1){:}f) \supset \bigvee \{i{:}\alpha \mid op(\alpha, Pre, Add, Del) \in Ops, f \in Add\} \in \Phi^n_\Pi$$

- **Conflict Exclusion Axioms:** for $i = 0, \ldots, n-1$:

$$\neg(i{:}\alpha_1 \wedge i{:}\alpha_2) \in \Phi^n_\Pi$$

for all $\alpha_1 \neq \alpha_2$ such that $op(\alpha_1, Pre_1, Add_1, Del_1) \in Ops$, $op(\alpha_2, Pre_2, Add_2, Del_2) \in Ops$, and $Pre_1 \cap Del_2 \neq \emptyset$ or $Pre_2 \cap Del_1 \neq \emptyset$.

It is immediate to see that the number of literals in Φ^n_Π is in $O(n|\mathcal{F}| + n|\mathcal{A}|)$. Moreover the number of Universal Axioms is in $O(nP_0|\mathcal{A}|)$ where P_0 is the maximal number of fluents mentioned in an operator (usually a small number); the number of Explanatory Frame Axioms is in $O(n|\mathcal{F}|)$; finally, the number of Conflict Exclusion Axioms is in $O(n|\mathcal{A}|^2)$.

3.2 Protocol Insecurity Problems as Planning Problems

Given a protocol insecurity problem $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$, it is possible to build a planning problem $\Pi_\Xi = \langle \mathcal{F}_\Xi, \mathcal{A}_\Xi, Ops_\Xi, I_\Xi, G_\Xi \rangle$ such that each solution to Π_Ξ can be translated back to a solution to Ξ: $\mathcal{F}_\Xi$ is the set of facts $\mathcal{S}$; $\mathcal{A}_\Xi$ and Ops_Ξ are the smallest sets such that $\ell\sigma \in \mathcal{A}_\Xi$ and $op(\ell\sigma, L\sigma, R\sigma \setminus L\sigma, L\sigma \setminus R\sigma) \in Ops$ for all $(L \xrightarrow{\ell} R) \in \mathcal{R}$ and all ground substitutions σ; finally $I_\Xi = \bigwedge\{f \mid f \in \mathcal{I}\} \bigwedge\{\neg f \mid f \in \mathcal{S}, f \notin \mathcal{I}\}$ and $G_\Xi = \bigvee_{S_\mathcal{B} \in \mathcal{B}} \bigwedge\{f \mid f \in S_\mathcal{B}\}$. For instance, the actions associated to (6) are of the form:

$$\begin{array}{l} op(step_1(A,B,C,K_{ab}), \\ \quad [w(0,A,A,[\,],[A,B,K_{ab}],C), \\ \quad\ fresh(nc(na,C))], \\ \quad [m(1,A,B,\{nc(C)\}_{K_{ab}}), \\ \quad\ w(2,B,A,[nc(na,C)],[A,B,K_{ab}],C)], \\ \quad [w(0,A,A,[\,],[A,B,K_{ab}],C), \\ \quad\ fresh(nc(na,C))]) \end{array}$$

The reduction of protocol insecurity problems to planning problems paves the way to an automatic SAT-compilation of protocol insecurity problems. However a direct application of the approach (namely the reduction of a protocol insecurity problem Ξ to a planning problem Π_Ξ followed by a SAT-compilation of Π_Ξ) is not immediately applicable. We therefore devised a set of optimizations and constraints whose combined effects often succeed in drastically reducing the size of the SAT instances.

3.3 Optimizations

Language specialization. We recall that for the reduction to propositional logic described in Section 3.1 to be applicable the set of fluents and actions must be finite. Unfortunately, the protocol insecurity problems introduced in Section 2 have an infinite number of facts and rule instances, and therefore the corresponding planning problems have an infinite number of fluents and actions. However the language can be restricted to a finite one since the set of states reachable in n steps is obviously finite (as long as the initial states comprise a finite number of facts). To determine a finite language capable to express the reachable states, it suffices to carry out a static analysis of the protocol insecurity problem.

To illustrate, let us consider again the simple protocol insecurity problem presented above and let $n = 7$, then $\|\texttt{int}\| = \{0,1,2\}$, $\|\texttt{user}\| = \{a,b\}$, $\|\texttt{iuser}\| = \{a,b,\texttt{intruder}\}$, $\|\texttt{key}\| = \{kab\}$, $\|\texttt{nonceid}\| = \{na\}$, $\|\texttt{session}\| = \bigcup_{i=0}^{\lfloor n/(k+1)\rfloor} s^i(\underline{1}) \bigcup_{i=0}^{\lfloor n/(k+1)\rfloor} s^i(\underline{2})$, where k is the number of protocol steps in a session run (in this case $k = 2$),[5] $\|\texttt{number}\| = nc(\texttt{nonceid},\texttt{session})$, $\|\texttt{func}\| = \bigcup_{i=0}^{n-1} f^i(\texttt{number})$, $\|\texttt{text}\| = \|\texttt{iuser}\| \cup \|\texttt{key}\| \cup \|\texttt{number}\| \cup \|\texttt{func}\| \cup \{\texttt{func}\}\texttt{key}$.[6]

Moreover, we can safely replace `list_of text` with $[\texttt{text},\texttt{text},\texttt{text}]$. The set of facts is then equal to $i(\texttt{text}) \cup fresh(\texttt{number}) \cup m(\texttt{int},\texttt{iuser},\texttt{iuser},\texttt{text}) \cup w(\texttt{int},\texttt{iuser},\texttt{user},\texttt{list\_of text},\texttt{list\_of text},\texttt{session})$ which consists of 10^{12} facts. This language is finite, but definitely too big for the practical applicability of the SAT encoding.

A closer look to the protocol reveals that the above language still contains many spurious facts. In particular the $m(\ldots)$, $w(\ldots)$, and $i(\cdot)$ can be specialized (e.g. by using specialized sorts to restrict the message terms to those messages which are allowed by the protocol). By analyzing carefully the facts of the form $m(\ldots)$ and $w(\ldots)$ occurring in the protocol rules of our exam-

[5] The bound on the number of steps implies a bound on the maximum number of possible session repetitions.

[6] If $S_1, \ldots, S_m$ and S' are sorts and f is a function symbol of arity $S_1, \ldots, S_m \to S'$, then $\|S_i\|$ is the set of terms of sort S_i and $f(S_1, \ldots, S_m)$ denotes $\{f(t_1, \ldots, t_m) \mid t_i \in \|S_i\|, i = 1, \ldots, m\}$.

ple we can restrict the sort `func` in such a way that $\|\texttt{func}\| = \{f(\texttt{number})\}$ and replace `list_of text` with $[\texttt{iuser}, \texttt{iuser}, \texttt{key}] \cup [\texttt{number}]$. Thanks to this optimization, the number of facts drops to $12,620$.

An other important language optimization borrowed from [15] splits message terms containing pairs of messages such as $m(j,s,r,\langle msg_1, msg_2\rangle)$ (where $\langle \_,\_\rangle$ is the pairing operator) into two message terms $m(j,s,r,msg_1,1)$ and $m(j,s,r,msg_2,2)$. (Due to the simplicity of the shown protocol, splitting messages has no impact on its language size.)

Fluent splitting. The second family of optimizations is based on the observation that in $w(j,s,r,ak,ik,c)$, the union of the first three arguments with the sixth form a key (in the data base theory sense) for the relation. This allows us to modify the language by replacing $w(j,s,r,ak,ik,c)$ with the conjunction of two new predicates, namely $wk(j,s,r,ak,c)$ and $inknw(j,s,r,ik,c)$. Similar considerations (based on the observation that the initial knowledge of a principal r does not depend on the protocol step j nor on principal s) allow us to simplify $inknw(j,s,r,ik,c)$ to $inknw(r,ik,c)$. Another effective improvement stems from the observation that ak and ik are lists. By using the set of new facts $wk(j,s,r,ak_1,1,c),\ldots,wk(j,s,r,ak_l,l,c)$ in place of $wk(j,s,r,[ak_1,\ldots,ak_l],c)$ the number of wk terms drops from $O(|\texttt{text}|^l)$ to $O(l|\texttt{text}|)$.[7] In the usual simple example the application of fluent splitting reduces the number of facts to $1,988$.

Exploiting static fluents. The previous optimization enables a new one. Since the initial knowledge of the honest principal does not change as the protocol execution makes progress, facts of the form $inknw(r,ik,c)$ occurring in the initial state are preserved in all the reachable states and those not occurring in the initial state will not be introduced. In the corresponding planning problem, this means that all the atoms $i : inknw(r,ik,c)$ can be replaced by $inknw(r,ik,c)$ for $i = 0,\ldots,n-1$ thereby reducing the number of propositional letters in the encoding. Moreover, since the initial state is unique, this transformation enables an off-line partial instantiation of the actions and therefore a simplification of the propositional formula.

Reducing the number of Conflict Exclusion Axioms. A critical issue in the propositional encoding technique described in Section 3.1 is the quadratic growth of the number of Conflict Exclusion Axioms in the number of actions. This fact often confines the applicability of the method to problems with a small number of actions. A way to lessen this difficulty is to reduce the number of conflicting axioms by considering the intruder knowledge as *monotonic*. Let f be a fact, S and S' be states, then we say that f is monotonic iff for all S if $f \in S$ and $S \to S'$, then $f \in S'$. Since a monotonic fluent never appears in the delete list of some action, then it cannot be a cause of a conflict. The idea here is to transform the rules so to make the facts of the form $i(\cdot)$ monotonic. The transformation on the rules is very simple as it amounts to adding the monotonic facts occurring in the left hand side of the rule to its right hand side. A consequence is that a monotonic fact simplifies the Explanatory Frame Axioms relative to it. The nice effect of this transformation is that the number of Conflict Exclusion Axioms generated by the associated planning problems drops dramatically.

Impersonate. The observation that most of the messages generated by the intruder by means of (12) and (13) are rejected by the receiver as non-expected or ill-formed suggests to restrict these rules so that the intruder sends only messages matching the patterns expected by the receiver. For each protocol rule of the form:

$$\ldots m(j,s,r,t) \cdot w(j,s,r,ak,ik,c) \cdot \ldots \xrightarrow{step_i(\ldots)} \ldots$$

we use a new rule of the form:

$$\begin{aligned} &\ldots \cdot w(j,s,r,ak,ik,c) \cdot i(s) \cdot i(r) \cdot i(t') \cdot \ldots \\ &\xrightarrow{impersonate_i(\ldots)} \ldots \cdot m(j,s,r,t') \cdot w(j,s,r,ak,ik,c) \\ &\qquad \cdot i(s) \cdot i(r) \cdot i(t') \cdot \ldots \end{aligned}$$

This rule states that if agent r is waiting for a message t from s and the intruder knows a term t' matching t, then the intruder can impersonate s and send t'. This optimization (borrowed from [16]) often reduces the number of rule instances in a dramatic way. In our example, this optimization step allows us to trade all the 1152 instances of (12) and (13) with 120 new rules.

It is easy to see that this transformation is correct as it preserves the existing attacks and does not introduce new ones.

Step compression. A very effective optimization, called *step compression* has been proposed in [9]. It consists of the idea of merging intruder with protocol rules. In particular, an impersonate rule:

$$\begin{aligned} &w(i,x_1,x_2,x_3,x_4,x_5) \cdot i(x_1) \cdot i(x_2) \cdot i(x_6) \\ &\qquad \xrightarrow{impersonate_i(\ldots)} m(i,x_1,x_2,x_6) \\ &\quad \cdot w(i,x_1,x_2,x_3,x_4,x_5) \cdot i(x_1) \cdot i(x_2) \cdot i(x_6) \end{aligned} \quad (14)$$

a generic protocol step rule:

$$\begin{aligned} &w(i,y_1,y_2,y_3,y_4,y_5) \cdot m(i,y_1,y_2,y_6) \xrightarrow{step_i(\ldots)} \\ &\quad w(j,y_1,y_2,y_7,y_4,y_5) \cdot m(i+1,y_2,y_1,y_8) \end{aligned} \quad (15)$$

[7] If S is a sort, then $|S|$ is the cardinality of $\|S\|$.

and a divert rule:

$$m(i+1,z_1,z_2,z_3) \xrightarrow{divert_{i+1}(\ldots)} i(z_1) \cdot i(z_2) \cdot i(z_3) \tag{16}$$

can be replaced by the following rule:

$$w(i,x_1,x_2,x_3,x_4,x_5)\sigma \cdot i(x_1)\sigma \cdot i(x_2)\sigma \cdot i(x_6)\sigma \xrightarrow{step\_comp_i(\ldots)\sigma} w(j,y_1,y_2,y_7,y_4,y_5)\sigma \cdot i(z_1) \cdot \sigma \cdot i(z_2)\sigma \cdot i(z_3)\sigma$$

where $\sigma = \sigma_1 \circ \sigma_2$ with $\sigma_1 = mgu(\{w(i,x_1,x_2,x_3,x_4,x_5) = w(i,y_1,y_2,y_3,y_4,y_5), m(i,x_1,x_2,x_6) = m(i,y_1,y_2,y_6)\})$ and $\sigma_2 = mgu(\{m(i+1,y_2,y_1,y_8) = m(i+1,z_1,z_2,z_3)\})$.

The rationale of this optimization is that we can safely restrict our attention to computation paths where (14), (15), and (16) are executed in this sequence without any interleaved action in between.

By applying this optimization we reduce both the number of facts (note that the facts of the form $m(\ldots)$ are no longer needed) and the number of rules as well as the number of steps necessary to find the attacks. For instance, by using this optimization the partial-order plan corresponding to the attack to the Needham-Schroeder Public Key (NSPK) protocol [21] has length 7 whereas if this optimization is disabled the length is 10, the numbers of facts decreases from 820 to 505, and the number of rules from 604 to 313.

3.4 Bounds and Constraints

In some cases in order to get encodings of reasonable size, we must supplement the above attack-preserving optimizations with the following bounding techniques and constraints. Even if by applying them we may loose some attacks, in our experience (cf. Section 4) this rarely occurs in practice.

Bounding the number of session runs. Let n and k be the bounds in the number of operation applications and in the number of protocol steps characterizing a protocol session respectively. Then the maximum number of times a session can be repeated is $\lfloor n/(k+1) \rfloor$. Our experience indicates that attacks usually require a number of session repetitions that is less than $\lfloor n/(k+1) \rfloor$. As a matter of fact two session repetitions are sufficient to find attacks to all the protocols we have analyzed so far. By using this optimization we can reduce the cardinality of the sort `session` (in the case of the NSPK protocol, we reduce it by a factor 1.5) and therefore the number of facts that depend on it.

Multiplicity of fresh terms. The number of fresh terms needed to find an attack is usually less than the number of fresh terms available. As a consequence, a lot of fresh terms allowed by the language associated with the protocol are not used, and many facts depending on them are allowed, but also not used. Often, one single fresh term for each fresh term identifier is sufficient for finding the attack. For instance the simple example shown above has the only fresh term identifier na and to use the only nonce $nc(na,\underline{1})$ is enough to detect the attack. Therefore, the basic idea of this constraint is to restrict the number of fresh terms available, thereby reducing the size of the language. For example, application of this constraint to the analysis of the NSPK protocol protocol preserves the detection of the attack and reduces the numbers of facts and rules from 313 to 87 and from 604 to 54 respectively. Notice that for some protocols such as the Andrew protocol [7] the multiplicity of fresh terms is necessary to detect the attack.

Constraining the rule variables. This constraint is best illustrated by considering the Kao-Chow protocol (see e.g. [7]):

(1) $A \rightarrow S : A, B, N_a$
(2) $S \rightarrow B : \{A, B, N_a, K_{ab}\}K_{as}, \{A, B, N_a, K_{ab}\}K_{bs}$
(3) $B \rightarrow A : \{A, B, N_a, K_{ab}\}K_{as}, \{N_a\}K_{ab}, N_b$
(4) $A \rightarrow B : \{N_b\}K_{ab}$

During the step (2) S sends B a pair of messages of which only the second component is accessible to B. Since B does not know K_{ab}, then B cannot check that the occurrence of A in the first component is equal to that inside the second. As a matter of fact, we might have different terms at those positions. The constraint amounts to imposing that the occurrences of A (as well as of B, N_a, and K_{ab}) in the first and in the second part of the message must coincide. Thus, messages of the form $\{a,b,nc(na,s(\underline{1}))\}kas, \{a,b,nc(nb,s(\underline{1}))\}kbs$ would be ruled out by the constraint. The application of this constraint allows us to get a feasible encoding of the Kao-Chow protocols in reasonable time. For instance, with this constraint disabled the encoding of the Kao Chow Repeated Authentication 1 requires more than 1 hour, otherwise it requires 16.34 seconds.

4 Implementation and Computer Experiments

We have implemented the above ideas in SATMC, a SAT-based Model-Checker for security protocol analysis. Given a protocol insecurity problem Ξ, a bound on the length of partial-order plan n, and a set of parameters specifying which bounds and constraints must be enabled (cf.

Section 3.4), SATMC first applies the optimizing transformations previously described to Ξ and obtains a new protocol insecurity problem Ξ', then Ξ' is translated into a corresponding planning problem $\Pi_{\Xi'}$ which is in turn compiled into SAT using the methodology outlined in Section 3.1. The propositional formula is then fed to a state-of-the-art SAT solver (currently Chaff [20], SIM [14], and SATO [24] are supported) and any model found by the solver is translated back into an attack which is reported to the user.

SATMC is one of the back-ends of the AVISS tool [2]. Using this tool, the user can specify a protocol and the security properties to be checked using a high-level specification language and the tool translates the specification in an Intermediate Format (IF) based on multiset rewriting. The notion of protocol insecurity problem given in this paper is inspired by the Intermediate Format. Some of the features supported by the IF (e.g. public and private keys, compound keys as well as other security properties such as authentication and secrecy) have been neglected in this paper for the lack of space. However, they are supported by SATMC.

We have run our tool against a selection of problems drawn from [7]. The results of our experiments are reported in Table 1 and they are obtained by applying all the previously described optimizations, by setting $n = 10$, by imposing two session runs for session, by allowing multiple fresh terms, and by constraining the rule variables. For each protocol we give the kind of the attack found (Attack), the number of propositional variables (Atoms) and clauses (Clauses), and the time spent to generate the SAT formula (EncT) as well as the time spent by Chaff to solve the corresponding SAT instance (SolveT). The label MO indicates a failure to analyze the protocol due to memory-out.[8] It is important to point out that for the experiments we found it convenient to disable the generation of Conflict Exclusion Axioms during the generation of the propositional encoding. Of course, by doing this, we are no longer guaranteed that the solutions found are linearizable and hence executable. SATMC therefore checks any partial order plan found for executability. Whenever a conflict is detected, a set of clauses excluding these conflicts are added to the propositional formula and the resulting formula is fed back to the SAT-solver. This procedure is repeated until an executable plan is found or no other models are found by the SAT solver. This heuristics reduces the size of the propositional encoding (it does not create the Conflicts Exclusion Axioms) and it can also reduce the computation time whenever the time required to perform the executability checks is less than the time required for generating the Conflict Exclusion Axioms. The experiments show that the SAT solving activity is carried out very quickly and that the overall time is dominated by the SAT encoding.

[8] Times have been obtained on a PC with a 1.4 GHz Processor and 512 MB of RAM. Due to a limitation of SICStus Prolog the SAT-based model-checker is bound to use 128 MB during the encoding generation.

5 Conclusions and Perspectives

We have proposed an approach to the translation of protocol insecurity problems into propositional logic based on the combination of a reduction to planning and well-known SAT-reduction techniques developed for planning. Moreover, we have introduced a set of optimizing transformations whose application to the input protocol insecurity problem drastically reduces the size of the corresponding propositional encoding. We have presented SATMC, a model-checker based on our ideas, and shown that attacks to a set of well-known authentication protocols are quickly found by state-of-the-art SAT solvers.

Since the time spent by SAT solver is largely dominated by the time needed to generate the propositional encoding, in the future we plan to keep working on ways to reduce the latter. A promising approach amounts to treating properties of cryptographic operations as invariants. Currently these properties are modeled as rewrite rules (cf. rule (10) in Section 2.1) and this has a bad impact on the size of the final encoding. A more natural way to deal with these properties amounts to building them into the encoding but this requires, among other things, a modification of the explanatory frame axioms and hence more work (both theoretical and implementational) is needed to exploit this very promising transformation.

Moreover, we would like to experiment SATMC against security problems with partially defined initial states. Problems of this kind occur when the initial knowledge of the principal in not completely defined or when the session instances are partially defined. We conjecture that neither the size of the SAT encoding nor the time spent by the SAT solver to check the SAT instances will be significantly affected by this generalization. But this requires some changes in the current implementation of SATMC and a thorough experimental analysis.

Bibliography

[1] Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, October 2001.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Moedersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocols Analysis Tool. In *14th International Conference on Computer-Aided Verification (CAV'02)*. 2002.

[3] David Basin and Grit Denker. Maude versus haskell: an experimental comparison in security protocol

Table 1: Performance of SATMC

Protocol	**Attack**	**Atoms**	**Clauses**	**EncT**	**SolveT**
ISO symmetric key 1-pass unilateral authentication	Replay	679	2,073	0.18	0.00
ISO symmetric key 2-pass mutual authentication	Replay	1,970	7,382	0.43	0.01
Andrew Secure RPC Protocol	Replay	161,615	2,506,889	80.57	2.65
ISO CCF 1-pass unilateral authentication	Replay	649	2,033	0.17	0.00
ISO CCF 2-pass mutual authentication	Replay	2,211	10,595	0.46	0.00
Needham-Schroeder Conventional Key	Replay STS	126,505	370,449	29.25	0.39
Woo-Lam Π	Parallel-session	7,988	56,744	3.31	0.04
Woo-Lam Mutual Authentication	Parallel-session	771,934	4,133,390	1,024.00	7.95
Needham-Schroeder Signature protocol	MM	17,867	59,911	3.77	0.05
Neuman Stubblebine repeated part	Replay STS	39,579	312,107	15.17	0.21
Kehne Langendorfer Schoenwalder (repeated part)	Parallel-session	-	-	MO	-
Kao Chow Repeated Authentication, 1	Replay STS	50,703	185,317	16.34	0.17
Kao Chow Repeated Authentication, 2	Replay STS	586,033	1,999,959	339.70	2.11
Kao Chow Repeated Authentication, 3	Replay STS	1,100,428	6,367,574	1,288.00	MO
ISO public key 1-pass unilateral authentication	Replay	1,161	3,835	0.32	0.00
ISO public key 2-pass mutual authentication	Replay	4,165	23,883	1.18	0.01
Needham-Schroeder Public Key	MM	9,318	47,474	1.77	0.05
Needham-Schroeder Public Key with key server	MM	11,339	67,056	4.29	0.04
SPLICE/AS Authentication Protocol	Replay	15,622	69,226	5.48	0.05
Encrypted Key Exchange	Parallel-session	121,868	1,500,317	75.39	1.78
Davis Swick Private Key Certificates, protocol 1	Replay	8,036	25,372	1.37	0.02
Davis Swick Private Key Certificates, protocol 2	Replay	12,123	47,149	2.68	0.03
Davis Swick Private Key Certificates, protocol 3	Replay	10,606	27,680	1.50	0.02
Davis Swick Private Key Certificates, protocol 4	Replay	27,757	96,482	8.18	0.13

Legenda: MM: Man-in-the-middle attack Replay STS: Replay attack based on a Short-Term Secret
MO: Memory Out

analysis. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[4] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 133–146. 1997.

[5] Common Authentication Protocol Specification Language. URL `http://www.csl.sri.com/~millen/capsl/`.

[6] Cervesato, Durgin, Mitchell, Lincoln, and Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[7] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL `http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[8] Ernie Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[9] Sebastian Moedersheim David Basin and Luca Viganò. An on-the-fly model-checker for security protocol analysis. forthcoming, 2002.

[10] Grit Denker, Jonathan Millen, and Harald Rueß. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, October 2000. Available at `http://www.csl.sri.com/~millen/capsl/`.

[11] Danny Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[12] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*. 1999.

[13] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1177. Morgan Kaufmann Publishers, San Francisco, 1997.

[14] Enrico Giunchiglia, Marco Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In Rajeev Goré, Aleander Leitsch, and Tobias Nipkow, editors, *Proceedings of IJCAR'2001*, LNAI 2083, pages 347–363. Springer-Verlag, Heidelberg, 2001.

[15] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

[16] Florent Jacquemard, Michael Rusinowitch, and Laurent Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer-Verlag, Heidelberg, 2000.

[17] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Morgan Kaufmann, San Francisco, California, 1996.

[18] Gawin Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998. See also `http://www.mcs.le.ac.uk/~gl7/Security/Casper/`.

[19] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996. See also `http://chacs.nrl.navy.mil/projects/crypto.html`.

[20] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*. 2001.

[21] R. M. (Roger Michael) Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. Technical Report CSL-78-4, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1978. Reprinted June 1982.

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

[23] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 192–202. IEEE Computer Society Press, 1999.

[24] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, Heidelberg, 1997.

Session III

Invited Talk

(joint with VERIFY)

Defining security is difficult and error prone

Dieter Gollmann
Microsoft Research
Cambridge, UK
diego@microsoft.com

Abstract

It is often claimed that the design of security protocols is difficult and error prone, and that surprising flaws are detected even after years of public scrutiny. A much quoted example is the Needham-Schroeder public key protocol published in 1978, where an attack was found as late as 1995. In consequence, it is concluded that the analysis of such protocols is too complex for analysis by hand and that the discipline (and tool support) of formal verification will have considerable impact on the design of security protocols.

The formal analysis that found the attack against the Needham-Schroeder public key protocol used correspondence properties to formally capture authentication. Tracing the history of correspondence to its origin one finds that this property was explicitly introduced to deal with the authentication of protocol runs and not with the authentication of a corresponding principal. Hence, one could argue that the attack formally violates an irrelevant property, and it is only by informal analysis that we can conclude that under certain circumstances (which are at odds with those assumed in the original paper) the original protocol goals are also violated. Incidentally, the attack is fairly simple and 'obvious' once one allows for 'dishonest' principals.

A second example are two logics proposed for SDSI name resolution. One of the logics allows derivations that have no correspondence in SDSI name resolution. This logic has a rule for keys 'speaking for' the same principal. The other logic is precise in the sense that the results of the name resolution algorithm exactly correspond to the derivations in the logic. This logic is key centric and does not relate keys issued to the same principal. Given that 'SDSI's groups provide simple, clear terminology for defining access control lists and security policies' it could be argued that the imprecise logic is actually better suited for the intended application of SDSI.

Finally, some recent work on protocol design for mobile communications will be sketched where a good part of the effort was devoted to deciding on the goals that actually should be achieved and on the assumptions about the environment the protocol was designed to run in.

As a common theme, our examples will stress that it is by no means straightforward to pick and formalize a security property, and that on occasion adopting 'standard' properties may mislead the verifier. Many security problems arise when a mechanism suited to one application is then re-deployed in a new environment, where either assumptions crucial for the original security argument no longer hold or where new security goals should be achieved. Thus, defining security is difficult and error prone and, unfortunately, verification of security protocols cannot simply proceed by picking from a menu of established properties and applying the latest tool but has to take care to justify the selection of security goals.

It is often claimed that the design of security protocols is difficult and error prone, with formal verification suggested as the recommended remedy. Verification requires a formal statement of the desired security properties and, maybe surprisingly, many protocols are broken simply by varying the assumptions on goals and intended environment. To defend my claim that defining security is difficult and error prone (and the really interesting challenge in formal verification) I will discuss some old and new examples of security protocols and their formal analysis.

Session IV

Verification of Security Protocols

(joint with VERIFY)

Identifying Potential Type Confusion in Authenticated Messages

Catherine Meadows

Code 5543
Naval Research Laboratory
Washington, DC 20375, USA
meadows@itd.nrl.navy.mil

Abstract

A *type confusion attack* is one in which a principal accepts data of one type as data of another. Although it has been shown by Heather et al. that there are simple formatting conventions that will guarantee that protocols are free from simple type confusions in which fields of one type are substituted for fields of another, it is not clear how well they defend against more complex attacks, or against attacks arising from interaction with protocols that are formatted according to different conventions. In this paper we show how type confusion attacks can arise in realistic situations even when the types are explicitly defined in at least some of the messages, using examples from our recent analysis of the Group Domain of Interpretation Protocol. We then develop a formal model of types that can capture potential ambiguity of type notation, and outline a procedure for determining whether or not the types of two messages can be confused. We also discuss some open issues.

1 Introduction

Type confusion attacks arise when it is possible to confuse a message containing data of one type with a message containing data of another. The most simple type confusion attacks are ones in which fields of one type are confused with fields of another type, such as is described in [7], but it is also possible to imagine attacks in which fields of one type are confused with a concatenation of fields of another type, as is described by Snekkenes in [8], or even attacks in which pieces of fields of one type are confused with pieces of fields of other types.

Simple type confusion attacks, in which a field of one type is confused with a field of another type, are easy to prevent by including type labels (tags) for all data and authenticating labels as well as data. This has been shown by Heather et al. [4], in which it is proved that, assuming a Dolev-Yao-type model of a cryptographic protocol and intruder, it is possible to prevent such simple type confusion attacks by the use of this technique. However, it is not been shown that this technique will work for more complex type confusion attacks, in which tags may be confused with data, and terms or pieces of terms of one type may be confused with concatenations of terms of several other types.[1] More importantly, though, although a tagging technique may work within a single protocol in which the technique is followed for all authenticated messages, it does not prevent type confusion of a protocol that uses the technique with a protocol that does not use the technique, but that does use the same authentication keys. Since it is not uncommon for master keys (especially public keys) to be used with more than one protocol, it may be necessary to develop other means for determining whether or not type confusion is possible. In this paper we explore these issues further, and describe a procedure for detecting the possibility of the more complex varieties of type confusion.

The remainder of this paper is organized as follows. In order to motivate our work, in Section Two, we give a brief account of a complex type confusion flaw that was recently found during an analysis of the Group Domain of Authentication Protocol, a secure multicast protocol being developed by the Internet Engineering Task Force. In Section Three we give a formal model for the use of types in protocols that takes into account possible type ambiguity. In Section Four we describe various techniques for constructing the artifacts that will be used in our procedure. In Section Five we give a procedure for determining whether it is possible to confuse the type of two messages. In Section Six we illustrate our procedure by showing how it could be applied to a simplified version of GDOI. In Section Seven we conclude the paper and give suggestions for further research.

2 The GDOI Attack

In this section we describe a type flaw attack that was found on an early version of the GDOI protocol.

The Group Domain of Interpretation protocol (GDOI)

[1] We believe that it could, however, if the type tags were augmented with tags giving the length of the tagged field, as is done in many implementations of cryptographic protocols.

[2], is a group key distribution protocol that is undergoing the IETF standardization process. It is built on top of the ISAKMP [6] and IKE [3] protocols for key management, which imposes some constraints on the way in which it is formatted. GDOI consists of two parts. In the first part, called the Groupkey Pull Protocol, a principal joins the group and gets a group key-encryption-key from the Group Controller/Key Distributor (GCKS) in a handshake protocol protected by a pairwise key that was originally exchanged using IKE. In the second part, called the Groupkey Push Message, the GCKS sends out new traffic encryption keys protected by the GCKS's digital signature and the key encryption key.

Both pieces of the protocol can make use of digital signatures. The Groupkey Pull Protocol offers the option of including a Proof-of-Possession field, in which either or both parties can prove possession of a public key by signing the concatenation of a nonce NA generated by the group member and a nonce NB generated by the GCKS. This can be used to show linkage with a certificate containing the public key, and hence the possession of any identity or privileges stored in that certificate.

As for the Groupkey Push Message, it is first signed by the GCKS's private key, and then encrypted with the key encryption key. The signed information includes a header HDR, (which is sent in the clear), and contains, besides the header, the following information:

1. a sequence number SEQ (to guard against replay attacks);
2. a security association SA;
3. a Key Download payload KD, and;
4. an optional certificate, CERT.

According to the conventions of ISAKMP, HDR must begin with a random or pseudo-random number. In pairwise protocols, this is jointly generated by both parties, but in GDOI, since the message must go from one to many, this is not practical. Thus, the number is generated by the GCKS. Similarly, it is likely that the Key Download message will end in a random number: a key. Thus, it is reasonable to assume that the signed part of a Groupkey Push Message both begins and ends in a random number.

We found two type confusion attacks. In both, we assume that the same private key is used by the GCKS to sign POPs and Groupkey Push Messages. In the first of these, we assume a dishonest group member who wants to pass off a signed POP from the GCKS as a Groupkey Push Message. To do this, we assume that she creates a fake plaintext Groupkey Push Message GPM, which is missing only the last (random) part of the Key Download Payload. She then initiates an instance of the Groupkey Pull Protocol with the GCKS, but in place of her nonce, she sends GPM. The GCKS responds by appending its nonce NB and signing it, to create a signed (GPM,NB). If NB is of the right size, this will look like a signed Groupkey Push Message. The group member can then encrypt it with the key encryption key (which she will know, being a group member) and send it out to the entire group.

The second attack requires a few more assumptions. We assume that there is a group member A who can also act as a GCKS, and that the pairwise key between A and another GCKS, B, is stolen, but that B's private key is still secure. Suppose that A, acting as a group member, initiates a Groupkey Pull Protocol with B. Since their pairwise key is stolen, it is possible for an intruder I to insert a fake nonce for B's nonce NB. The nonce he inserts is a fake Groupkey Push Message GPM' that it is complete except for a prefix of the header consisting of all or part of the random number beginning the header. A then signs (NA,GPM'), which, if NA is of the right length, will look like the signed part of a Groupkey Push Message. The intruder can then find out the key encryption key from the completed Groupkey Pull Protocol and use it to encrypt the resulting (NA,GPM') to create a convincing fake Groupkey Push Message.

Fortunately, the fix was simple. Although GDOI was constrained by the formatting required by ISAKMP, this was not the case for the information that was signed within GDOI. Thus, the protocol was modified so that, whenever a message was signed within GDOI, information was prepended saying what the purpose was (e.g. a member's POP, or a Groupkey Push Message). This eliminated the type confusion attacks.

There are several things to note here. The first is that existing protocol analysis tools are not very good at finding these types of attacks. Most assume that some sort of strong typing is already implemented. Even when this is not the case, the ability to handle the various combinations that arise is somewhat limited. For example, we found the second, less feasible, attack automatically with the NRL Protocol Analyzer, but the tool could not have found the first attack, since the ability to model it requires the ability to model the associativity of concatenation, which the NRL Protocol Analyzer lacks. Moreover, type confusion attacks do not require a perfect matching between fields of different types. For example, in order for the second attack to succeed, it is not necessary for NA to be the same size as the random number beginning the header, only that it be no longer than that number. Again, this is something that is not within the capacity of most crypto protocol analysis tools. Finally, most crypto protocol analysis tools are not equipped for probabilistic analysis, so they would not be able to find cases in which, although type confusion would not be possible every time, it would occur with a high enough probability to be a concern.

The other thing to note is that, as we said before, even though it is possible to construct techniques that can be used to guarantee that protocols will not interact insecurely with other protocols that are formatted using the same

technique, it does not mean that they will not interact insecurely with protocols that were formatted using different techniques, especially if, in the case of GDOI's use of ISAKMP, the protocol wound up being used differently than it was originally intended (for one-to-many instead of pairwise communication). Indeed, this is the result one would expect given previous results on protocol interaction [5, 1]. Since it is to be expected that different protocols will often use the same keys, it seems prudent to investigate to what extent an authenticated message from one protocol could be confused with an authenticated message from another, and to what extent this could be exploited by a hostile intruder. The rest of this paper will be devoted to the discussion of a procedure for doing so.

3 The Model

In this section we will describe the model that underlies our procedure. It is motivated by the fact that different principals may have different capacities for checking types of messages and fields in messages. Some information, like the length of the field, may be checkable by anybody. Other information, like whether or not a field is a random number generated by a principal, or a secret key belonging to a principal, will only be checkable by the principal who generated the random number in the first case, and by the possessor(s) of the secret key in the second place. In order to do this, we need to develop a theory of types that take differing capacities for checking types into account.

We assume an environment consisting of principals who possess information and can check properties of data based on that information. Some information is public and is shared by all principals. Other information may belong to only one or a few principals.

Definition 3.1 *A* field *is a sequence of bits. We let ι denote the empty field. If x and y are two fields, we let $x||y$ denote the concatenation of x and y. If $\bar{x}$ and $\bar{y}$ are two lists of fields, then we let $append(\bar{x}, \bar{y})$ denote the list obtained by appending $\bar{y}$ to $\bar{x}$.*

Definition 3.2 *A* type *is a set of fields, which may or may not have a probability distribution attached. If P is a principal, then a* type local to P *is a type such that membership in that type is checkable by P. A* public type *is one whose membership is checkable by all principals. If G is a group of principals, then a* type private to G *is a type such that membership in that type is checkable by the members of G and only the members of G.*

Examples of a public type would be all strings of length 256, the string "key," or well-formed IP addresses. Examples of private types would be a random nonce generated by a principal (private to that principal) a principal's private signature key (private to that principal), and a secret key shared by Alice and Bob (private to Alice and Bob, and perhaps the server that generated the key, if one exists). Note that a private type is not necessarily secret; all that is required is that only members of the group to whom the type is private have a guaranteed means of checking whether or not a field belongs to that type. As in the case of the random number generated by a principal, other principals may have been told that a field belongs to the type, but they do not have a reliable means of verifying this.

The decision as to whether or not a type is private or public may also depend upon the protocol in which it is used and the properties that are being proved about the protocol. For example, to verify the security of a protocol that uses public keys to distribute master keys, we may want to assume that a principal's public key is a public type, while if the purpose of the protocol is to validate a principal's public key, we may want to assume that the type is private to that principal and some certification authority. If the purpose of the protocol is to distribute the public key to the principal, we may want to assume that the type is private to the certification authority alone.

Our use of public and local types is motivated as follows. Suppose that an intruder wants to fool Bob into accepting an authenticated message M from a principal Alice as an authenticated message N from Alice. Since M is generated by Alice, it will consist of types local to her. Thus, for example, if M is supposed to contain a field generated by Alice it will be a field generated by her, but if it is supposed to contain a field generated by another party, Alice may only be able to check the publically available information such as the formatting of that field before deciding to include it in the message. Likewise, if Bob is verifying a message purporting to be N, he will only be able to check for the types local to himself. Thus, our goal is to be able to check whether or not a message built from types local to Alice can be confused with another message built from types local to Bob, and from there, to determine if an intruder is able to take advantage of this to fool Bob into producing a message that can masquerade as one from Alice.

We do not attempt to give a complete model of an intruder in this paper, but we do need to have at at least some idea of what types mean from the point of view of the intruder to help us in computing the probability of an intruder's producing type confusion attacks. In particular, we want to determine the probability that the intruder can produce (or force the protocol to produce) a field of one type that also belongs to another type. Essentially, there are two questions of interest to an intruder: given a type, can it control what field of that type is sent in a message, and given a type, will any arbitrary member of that type be accepted by a principal, or will a member be accepted only with a certain probability.

Definition 3.3 *We say that a type is* under the control of the intruder *if there is no probability distribution associ-*

ated with it. We say that a type is probabilistic *if there a a probability distribution associated with it. We say that a probabilistic type local to a principal A is* under the control of *A if the probability of A accepting a field as a member of X is given by the probability distribution associated with X.*

The idea behind probabilistic types and types under control of the intruder is that the intruder can choose what member of a type can be used in a message if it is under its control, but for probabilistic types the field used will be chosen according to the probability distribution associated with the type. On the other hand, if a type is not under the control of a principal A, then A will accept any member of that type, while if the type is under the control of A, she will only accept an element as being a member of that type according to the probability associated with that type.

An example of a type under the control of an intruder would be a nonce generated by the intruder, perhaps while impersonating someone else. An example of a probabilistic type that is not under the control of A would be a nonce generated by another principal B and sent to A in a message. An example of a probabilistic type that is also under the control of A would be a nonce generated by A and sent by A in a message, or received by A in some later message.

Definition 3.4 *Let X and Y be two types. We say that $X \sqcap Y$ holds if an intruder can force a protocol to produce an element x of X that is also an element of Y.*

Of course, we are actually interested in the probability that $X \sqcap Y$ holds. Although the means for calculating $P(X \sqcap Y)$ may vary, we note that the following holds if there are no other constraints on X and Y:

1. If X and Y are both under the control of the intruder, then $P(X \sqcap Y)$ is 1 if $X \cap Y \neq \phi$ and is zero otherwise;
2. If X is under the control of the intruder, and Y is a type under the control of A, and the intruder knows the value of the member of Y before choosing the member of X, then $P(Y \sqcap X) = P(\hat{x} \in X \cap Y)$, where $\hat{x}$ is the random variable associated with X;
3. If X a type under the control of A, and Y is a type local to B but not under the control of B, then $P(X \sqcap Y) = P(\hat{x} \in X \cap Y)$;
4. If X is under the control of A and Y is under the control of some other (non-intruder) B, then $P(Y \sqcap X) = P(\hat{x} = \hat{y})$ where $\hat{x}$ is the random variable associated with X, and $\hat{y}$ is the random variable associated with Y.

Now that we have a notion of type for fields, we extend it to a notion of type for messages.

Definition 3.5 *A* message *is a concatenation of one or more fields.*

Definition 3.6 *A* message type *is a function $\mathcal{R}$ from lists of fields to types, such that:*

1. *The empty list is in $\mathrm{Dom}(\mathcal{R})$;*
2. *$\langle x_1, ..., x_k \rangle \in \mathrm{Dom}(\mathcal{R})$ if and only if $\langle x_1, ..., x_{k-1} \rangle \in \mathrm{Dom}(\mathcal{R})$ and $x_k \in \mathcal{R}(\langle x_1, ..., x_{k-1} \rangle)$;*
3. *If $\langle x_1, ..., x_k \rangle \in \mathrm{Dom}(\mathcal{R})$, and $x_k = \iota$, then $\mathcal{R}(\langle x_1, ..., x_k \rangle) = \{\iota\}$, and ;*
4. *For any infinite sequence $S = \langle ..., x_i, ... \rangle$ such that all prefixes of S are in $Dom(\mathcal{R})$, there exists an n such that, for all $i > n$, $x_i = \iota$.*

The second part of the definition shows how, once the first $k-1$ fields of a message are known, then $\mathcal{R}$ can be used to predict the type of the k'th field. The third and fourth parts describe the use of the empty list ι in indicating message termination. The third part says that, if the message terminates, then it can't start up again. The fourth part says that all messages must be finite. Note, however, that it does not require that messages be of bounded length. Thus, for example, it would be possible to specify, say, a message type that consists of an unbounded list of keys.

The idea behind this definition is that the type of the n'th field of a message may depend on information that has gone before, but exactly where this information goes may depend upon the exact encoding system used. For example, in the tagging system in [4], the type is given by a tag that precedes the field. In many implementations, the tag will consist of two terms, one giving the general type (e.g. "nonce"), and the other giving the length of the field. Other implementations may use this same two-part tag, but it may not appear right before the field; for example in ISAKMP, and hence in GDOI, the tag refers, not to the field immediately following it, but the field immediately after that. However, no matter how tagging is implemented, we believe that it is safe to assume that any information about the type of a field will come somewhere before the field, since otherwise it might require knowledge about the field that only the tag can supply (such as where the field ends) in order to find the tag.

Definition 3.7 *The* support *of a message type $\mathcal{R}$ is the set of all messages of the form $x_1||...||x_n$ such that $\langle x_1, ..., x_n \rangle \in \mathrm{Dom}(\mathcal{R})$.*

For an example of a message type, we consider a message of the form

$$\text{"}nonce\text{''}, N_1, NONCE_1, \text{"}nonce\text{''}, N_2, NONCE_2$$

where $NONCE_1$ is a random number of length N_1 generated by the creator of the message, N_1 is a 16-bit integer, and $NONCE_2$ is a random number of length N_2, where

both $NONCE_2$ and N_2 are generated by the intended receiver, and N_2 is another 16-bit integer. From the point of view of the generator of the message, the message type is as follows:

1. $\mathcal{R}(\langle\rangle) = \text{“}nonce''$.
2. $\mathcal{R}(\langle\text{“}nonce''\rangle) = \{X|length(X) = 16\}$. Since N_1 is generated by the sender, it is a type under the control of the sender consisting of the set of 16-bit integers, with a certain probability attached.
3. $\mathcal{R}(\langle\text{“}nonce'', N_1\rangle) = \{X|length(X) = N_1\}$. Again, this is a private type consisting of the set of fields of length N_1. In this case, we can choose the probability distribution to be the uniform one.
4. $\mathcal{R}(\langle\text{“}nonce'', N_1, NONCE_1\rangle) = \{\text{“}nonce''\}$.
5. $\mathcal{R}(\langle\text{“}nonce'', N_1, NONCE_1, \text{“}nonce''\rangle) = \{X|length(X) = 16\}$. Since the sender did not actually generate N_2, all he can do is check that it is of the proper length, 16. Thus, this type is not under the control of the sender. If N_2 was not authenticated, then it is under the control of the intruder.
6. $\mathcal{R}(\langle\text{“}nonce'', N_1, NONCE_1, \text{“}nonce'', N_2\rangle) = \{Y|length(Y) = N_2\}$. Again, this value is not under the control of the sender, all the principal can do is check that what purports to be a nonce is indeed of the appropriate length.
7. $\mathcal{R}(\langle\text{“}nonce'', N_1, NONCE_1, \text{“}nonce'', N_2, NONCE_1, \rangle) = \{\iota\}$. This last tells us that the message ends here.

From the point of view of the receiver of the message, the message type will be somewhat different. The last two fields, N_2 and $NONCE_2$ will be types under the control of the receiver, while N_1 and $NONCE_1$ will be types not under its control, and perhaps under the control of the intruder, whose only checkable property is their length. This motivates the following definition:

Definition 3.8 *A* message type local to a principal P *is a message type $\mathcal{R}$ whose range is made up of types local to P.*

We are now in a position to define type confusion.

Definition 3.9 *Let $\mathcal{R}$ and $\mathcal{S}$ be two message types. We say that a pair of sequences $\langle x_1, ..., x_n\rangle \in Dom(\mathcal{R})$ and $\langle y_1, ..., y_n\rangle \in Dom(\mathcal{S})$ is a* type confusion between $\mathcal{R}$ and $\mathcal{S}$ *if:*

1. $\iota \in \mathcal{R}(\langle x_1, ..., x_n\rangle)$*;*
2. $\iota \in \mathcal{S}(\langle y_1, ..., y_m\rangle)$*, and;*
3. $x_1||...||x_n = y_1||...||y_m$.

The first two conditions say that the sequences describe complete messages. That last conditions says that the messages, considered as bit-strings, are identical.

Definition 3.10 *Let $\mathcal{R}$ and $\mathcal{S}$ be two message types. We say that $\mathcal{R} \sqcap \mathcal{S}$ holds if an intruder is able to force a protocol to produce an $\bar{x}$ in $Dom(\mathcal{R})$ such that there exists $\bar{y}$ in $Dom(\mathcal{S})$ such that $(\bar{x}, \bar{y})$ is a type confusion..*

Again, what we are interested in is computing, or at least estimating, $P(\mathcal{R} \sqcap \mathcal{S})$. This will be done in Section 5.

4 Constructing and Rearranging Message Types

In order to perform our comparison procedure, we will need the ability to build up and tear down message types, and create new message types out of old. In this section we describe the various ways that we can do this.

We begin by defining functions that are restrictions of message types (in particular to prefixes and postfixes of tuples).

Definition 4.1 *An* n-postfix message type *is a function $\mathcal{R}$ from tuples of length n or greater to types such that:*

1. *For all $k > 0$, $\langle x_1, ..., x_{n+k}\rangle \in \mathrm{Dom}(\mathcal{R})$ if and only if $x_{n+k} \in \mathcal{R}(\langle x_1, ..., x_{n+k-1}\rangle)$;*
2. *If $\langle x_1, ..., x_{n+k}\rangle \in \mathrm{Dom}(\mathcal{R})$, and $x_{n+k} = \iota$, then $\mathcal{R}(\langle x_1, ..., x_{n+k+1}\rangle) = \{\iota\}$, and ;*
3. *For any infinite sequence $S = \langle ..., x_i, ...\rangle$ such that all prefixes of S of length n and greater are in $Dom(\mathcal{R})$, there exists an m such that, for all $i > m$, $x_i = \iota$.*

We note that the restriction of a message type $\mathcal{R}$ to sequences of length n or greater is an n-postfix message type, and that a message type is a 0-postfix message type.

Definition 4.2 *An n-prefix message type is a function $\mathcal{R}$ from tuples of length less than n to types such that:*

1. *$\mathcal{R}$ is defined over the empty list;*
2. *For all $k < n$, $\langle x_1, ..., x_k\rangle \in \mathrm{Dom}(\mathcal{R})$ if and only if $x_k \in \mathcal{R}(\langle x_1, ..., x_{k-1}\rangle)$, and;*
3. *If $k < n - 1$, and $\langle x_1, ..., x_k\rangle \in \mathrm{Dom}(\mathcal{R})$, and $x_k = \iota$, then $\mathcal{R}(\langle x_1, ..., x_{k+1}\rangle) = \{\iota\}$.*

We note that the restriction of a message type to sequences of length less than n is an n-prefix message type.

Definition 4.3 *We say that a message type or n-prefix message type $\mathcal{R}$ is* t-bounded *if $\mathcal{R}(x) = \iota$ for all tuples x of length t or greater.*

In particular, a message type that is both t-bounded and t-postfix will be a trivial message type.

Definition 4.4 *Let $\mathcal{R}$ be an n-postfix message type. Let X be a set of m-tuples in the pre-image of $\mathcal{R}$, where m $\geq$ n. Then $\mathcal{R}\lfloor X$ is defined to be the restriction of R to the set of all $\langle x_1, ..., x_m, ..., x_r\rangle$ in $Dom(\mathcal{R})$ such that $\langle x_1, ..., x_m\rangle \in X$.*

Definition 4.5 *Let $\mathcal{R}$ be an n-prefix message type. Let X be a set of n-1 tuples. Then $\mathcal{R}\lceil X$ is defined to be the restriction of $\mathcal{R}$ to the set of all tuples $\bar{x}$ such that $\bar{x} \in X$, or $\bar{x} = \langle x_1, ...x_i\rangle$ such that there exists $\langle y_{i+1}, ..., y_{n-1}\rangle$ such that $\langle x_1, ...x_i, y_{i+1}, ..., y_{n-1}\rangle \in X$.*

Definition 4.6 *Let $\mathcal{R}$ be an n-postfix message type. Then $Split(\mathcal{R})$ is the function whose domain is the set of all $\langle x_1, ..., x_n, y_1, y_2, x_{n+2}, ..., x_m\rangle$ of length n+1 or greater such that $\langle x_1, ..., x_n, y_1||y_2, x_{n+2}, ..., x_m\rangle \in Dom(\mathcal{R})$ and such that*

a. For the tuples of length $i > n$ +1, $Split(\mathcal{R})(\langle x_1, ..., x_n, y_1, y_2, x_{n+2}, ..., X_m\rangle) = \mathcal{R}(\langle x_1, ..., x_n, y_1||y_2, x_{n+2}, ..., x_m\rangle)$, and;

b. For tuples of length n +1 , $Split(\mathcal{R})(\langle y_1, ..., y_{n+1}\rangle) = \{z \mid \langle y_1, ..., y_{n+1}||z\rangle \in Dom(\mathcal{R})$.

Definition 4.7 *Let $\mathcal{R}$ be an n-prefix message type. Let F be a function from a set of n-tuples to types such that there is at least one tuple $\langle x_{i+1}..., x_n\rangle$ in the domain of F such that $\langle x_{i+1}..., x_{n-1}\rangle$ is in the domain of $\mathcal{R}$. Then $\mathcal{R}\sharp F$, the* extension of $\mathcal{R}$ by F, *is the function whose domain is*

a. For $i < n$, the set of all $\langle x_1...., x_i\rangle$ such that $\langle x_1...., x_i\rangle \in Dom(\mathcal{R})$, and such that there exists $\langle x_{i+1}..., x_n\rangle$ such that $\langle x_1...., x_i, x_{i+1}..., x_n\rangle \in Dom(F)$;

b For i = n, the set of all $\langle x_1...., x_{n-1}, x_n\rangle$ such that $\langle x_1...., x_{n-1}\rangle \in Dom(\mathcal{R})$ and $\langle x_1...., x_{n-1}, x_n\rangle \in Dom(F)$;

and whose restriction to tuples of length less than n is $\mathcal{R}$, and whose restriction to n-tuples is F.

Proposition 4.1 *If $\mathcal{R}$ is an n-postfix message type, then $\mathcal{R}\lfloor X$ is an m-postfix message type for any set of m-tuples X, and $Split(\mathcal{R})$ is an (n+1)-postfix message type. If $\mathcal{R}$ is t-bounded, then so is $\mathcal{R}\lfloor X$, while $Split(\mathcal{R})$ is (t+1)-bounded. Moreover, if S is an n-prefix message type, then so is $\mathcal{S}\lceil Y$ for any set of n-1 tuples Y, and $\mathcal{S}\sharp F$ is an (n+1)-prefix message type for any function F from n-tuples to types such at for at least one element $\langle x_{i+1}..., x_n\rangle$ in the domain of F, $\langle x_{i+1}..., x_{n-1}\rangle$ is in the domain of $\mathcal{S}$.*

We close with one final definition.

Definition 4.8 *Let F be a function from k-tuples of fields to types. We define $Pre(F)$ to be the function from k-tuples of fields to types defined by $Pre(F)(x)$ is the set of all prefixes of all elements of $F(x)$.*

5 The Zipper: A Procedure for Comparing Message Types

We now can define our procedure for determining whether or not type confusion is possible between two message types $\mathcal{R}$ and $\mathcal{S}$, that is, whether it is possible for a verifier to mistake a message of type $\mathcal{R}$ generated by some principal for a message of type $\mathcal{S}$ generated by that same principal , where $\mathcal{R}$ is a message type local to the generator, and $\mathcal{S}$ is a message type local to the verifier. But, in order for this to occur, the probability of $\mathcal{R} \sqcap \mathcal{S}$ must be nontrivial. For example, consider a case in which $\mathcal{R}$ is a type local to and under the control of Alice consisting of a random variable 64 bits long, and $\mathcal{S}$ consists of another random 64-bit variable local to and under the control of Bob. It is possible that $\mathcal{R} \sqcap \mathcal{S}$ holds, but the probability that this is so is only $1/2^{64}$. On the other hand, if $\mathcal{R}$ is under the control of the intruder, then the probability that their support is non-empty is one. Thus, we need to choose a threshold probability, such that we consider a type confusion whose probability falls below the threshold to be of negligible consequence.

Once we have chosen a threshold probability, our strategy will be to construct a "zipper"between the two message types to determine their common support. We will begin by finding the first type of $\mathcal{R}$ and the first type of $\mathcal{S}$, and look for their intersection. Once we have done this, for each element in the common support, we will look for the intersection of the next two possible types of $\mathcal{R}$ and $\mathcal{S}$, respectively, and so on. Our search will be complicated, however, by the fact that the matchup may not be between types, but between pieces of types. Thus, for example, elements of the first type of $\mathcal{R}$ may be identical to the prefixes of elements of the first type of $\mathcal{S}$, while the remainders of these elements may be identical to elements of the second type of $\mathcal{R}$, and so forth. So we will need to take into account three cases: the first, where two types have a nonempty intersection, the second, where a type from $\mathcal{R}$ (or a set of remainders of types from $\mathcal{R}$) has a nonempty intersection with a set of prefixes from the second type of $\mathcal{S}$, and the third, where a type from $\mathcal{S}$ (or a set of remainders of types from $\mathcal{S}$) has a nonempty intersection with a set of prefixes from the second type of $\mathcal{R}$. All of these will impose a constraint on the relative lengths of the elements of the types from $\mathcal{S}$ and $\mathcal{R}$, which need to be taken into account, since some conditions on lengths may be more likely to be satisfied than others.

Our plan is to construct our zipper by use of a tree in which each node has up to three possible child nodes, corresponding to the three possibilities given above. Let $\mathcal{R}$ and $\mathcal{S}$ be two message types, and let p be a number between 1 and 0, such that we are attempting to determine whether the probability of constructing a type confusion between $\mathcal{R}$ and $\mathcal{S}$ is greater than p. We define a tertiary tree of sept-tuples as follows. The first entry of each sept-tuple is a set U of triples $\langle x, \bar{y}, \bar{z}\rangle$, where x is a bitstring and $\bar{y} = \langle y_1, ..., y_n\rangle$ and $\bar{z} = \langle z_1, ..., z_m\rangle$ such that $y1||...||y_n = z_1||...||z_m = x$. We will call U the *support* of the node. The second and third entries are n and m postfix message types, respectively. The fourth and fifth are message types or prefix message types. The sixth is a probability q. The seventh is a set of constraints on lengths of types. The root of the tree is of the form $\langle \phi, \mathcal{R}, \mathcal{S}, \langle\rangle, \langle\rangle, 1, D\rangle$, where D is the set of length constraints introduced by $\mathcal{R}$ and $\mathcal{S}$.

Given a node, $\langle U, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, q, C\rangle$, we construct up to three child nodes as follows:

1. The first node corresponds to the case in which a term from H can be confused with a term from I. Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle \in U$ such that $P(\mathcal{H}(\bar{y}) \cap \mathcal{I}(\bar{z}) \neq \phi) \cdot q > p$. Then, if T is non-empty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in \mathcal{H}_n(\bar{y})$, $\bar{y}' = append(\bar{y}, \langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle)$;
   Note that, by definition y_1 is an element of $\mathcal{I}(\bar{z})$ as well as $\mathcal{H}(\bar{y})$.

   b. The second element is the (n+1)-postfix message type $\mathcal{H}\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the (m+1)-postfix message type $\mathcal{I}\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp\mathcal{H}_n)\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp\mathcal{I}_m)\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   f. The sixth element is $max(\{P(\mathcal{H}_n(\bar{y}) \cap \mathcal{I}_m(\bar{z}) \neq \phi \mid \exists x s.t.(x, \bar{y}, \bar{z}) \in T)\}) \cdot q$, and;

   g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) = \text{length}(\mathcal{I}_n)$.

   We call this first node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) = \text{length}(\mathcal{I}_m)$.

2. The second node corresponds to the case in which a type from $\mathcal{H}$ can be confused wit prefix of a type from $\mathcal{I}$.

   Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle$ such that $P(\mathcal{H}_n(\bar{y}) \sqcap Pre(\mathcal{I}_m)(\bar{z})) \cdot q > p$. Then, if T is non-empty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in \mathcal{H}_n(\bar{y})$, $\bar{y}' = append(\bar{y}.\langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle)$;
   Note that, in this case y_1 is an element of $Pre(\mathcal{I}_m)(\bar{z}))$ as well.

   b. The second element is the (n+1)-postfix message type $\mathcal{H}\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the m-postfix message type $Split(\mathcal{I})\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp\mathcal{H}_n)\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp Pre(\mathcal{I}_m))\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   f. The sixth element of the tuple is $max(\{P(\mathcal{H}_n(\bar{y}) \sqcap Pre(\mathcal{I}_m)(\bar{z}) \mid \exists x s.t.(x, \bar{y}, \bar{z}) \in T))\}) \cdot q$, and;

   g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) < \text{length}(\mathcal{I}_m)$.

   We call this node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) < \text{length}(\mathcal{I}_m)$.

3. The third node corresponds to the case in which a prefix of a type from $\mathcal{H}$ can be confused with a type from $\mathcal{I}$.

   Let T be the set of all $\langle x, \bar{y}, \bar{z}\rangle$ in Usuch that $P(Pre(\mathcal{H}_n)(\bar{y}) \sqcap \mathcal{I}(\bar{z})) \cdot q > p$. Then, if T is nonempty, we construct a child node as follows:

   a. The first element of the new tuple is the set T' of all $\langle x', \bar{y}', \bar{z}'\rangle$ such that there exists $\langle x, \bar{y}, \bar{z}\rangle \in T$ such that $x' = x||y_1$, where $y_1 \in Pre(\mathcal{H}_n)(\bar{y})$, $\bar{y}' = append(\bar{y}, \langle y_1\rangle)$, and $\bar{z}' = append(\bar{z}, \langle y_1\rangle$;
   Note that, in this case y_1 is an element $\mathcal{I}_m(\bar{z})$ as well.

   b. The second element is the n-postfix message type $Split(\mathcal{H})\lfloor W_R$, where $W_R = \{\bar{y}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   c. The third element is the (m+1)-postfix message type $\mathcal{I}\lfloor W_S$, where $W_S = \{\bar{z}' | \langle x', \bar{y}', \bar{z}'\rangle \in T'\}$;

   d. The fourth element is $(\mathcal{J}\sharp Pre(\mathcal{H}_n)))\lceil V_R$, where $V_R = \{\bar{y} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

   e. The fifth element is $(\mathcal{K}\sharp\mathcal{I}_m)\lceil V_S$, where $V_S = \{\bar{z} | \langle x, \bar{y}, \bar{z}\rangle \in T\}$;

f. The sixth element is $max(\{P(Pre(\mathcal{H}_n)(\bar{y}) \sqcap \mathcal{I}_m(\bar{z})) \mid \exists x s.t.(x, \bar{y}, \bar{z}) \in T)\}) \cdot q$, and;

g. The seventh element is $C \cup \{c_1\}$, where c_1 is the constraint $\text{length}(\mathcal{H}_n) > \text{length}(\mathcal{I}_m)$.

We call this node the *node generated by the constraint* $\text{length}(\mathcal{H}_n) > \text{length}(\mathcal{I}_m)$.

The idea behind the nodes in the tree is as follows. The first entry in the sept-tuple corresponds to the part of the zipper that we have found so far. The second and third corresponds to the portions of $\mathcal{R}$ and $\mathcal{S}$ that are still to be compared. The fourth and fifth correspond to the portions of $\mathcal{R}$ and $\mathcal{S}$ that we have compared so far. The sixth entry gives an upper bound on the probability that this portion of the zipper can be constructed by an attacker. The seventh entry gives the constraints on lengths of fields that are satisfied by this portion of the zipper.

Definition 5.1 *We say that a zipper* succeeds *if it contains a node* $\langle U, \langle\rangle, \langle\rangle, \mathcal{J}, \mathcal{K}, q, C\rangle$.

Theorem 5.1 *The zipper terminates for bounded message types, and, whether or not it terminates, it succeeds if there are any type confusions of probability greater than p. For bounded message types, the complexity is exponential in the number of message fields.*

6 An Example: An Analysis of GDOI

In this section we give a partial analysis of the signed messages of a simplified version of the GDOI protocol.

There are actually three such messages. They are: the POP signed by the group member, the POP signed by the GCKS, and the Groupkey Push Message signed by the GCKS. We will show how the POP signed by the GCKS can be confused with the Groupkey Push Message.

The POP is of the form $NONCE_A, NONCE_B$ where $NONCE_A$ is a random number generated by a group member, and $NONCE_B$ is a random number generated by the GCKS. The lengths of $NONCE_A$ and $NONCE_B$ are not constrained by the protocol. Since we are interested in the types local to the GCKS, we have $NONCE_A$ the type consisting of all numbers, and $NONCE_B$ the type local to the GCKS consisting of the the single nonce generated by the GCKS.

We can thus define the POP as a message type local to the GCKS as follows:

1. $\mathcal{R}(\langle\rangle) = NONCE_A$ where $NONCE_A$ is the type under the control of the intruder consisting of all numbers, and;
2. $\mathcal{R}(\langle y_1\rangle) = NONCE_B$ where $NONCE_B$ is a type under control of the GCKS.

We next give a simplified (for the purpose of exposition) Groupkey Push Message. We describe a version that consists only of the Header and the Key Download Payload:

$NONCE_H$, kd, $MESSAGE\_LENGTH$, sig, $KDLENGTH, KDHEADER, KEYS$

The $NONCE_H$ at the beginning of the header is of fixed length (16 bytes). The one-byte kd field gives the type of the first payload, while the 4-byte $MESSAGE\_LENGTH$ gives the length of the message in bytes. The one-byte sig field gives the type of the next payload (in this case the signature, which is not part of the signed message), while the 2-byte $KDLENGTH$ gives the length of the key download payload. We divide the key download data into two parts, a header which gives information about the keys, and the key data, which is random and controlled by the GCKS. (This last is greatly simplified from the actual GDOI specification).

We can thus define the Groupkey Push Message as the following message type local to the intended receiver:

1. $\mathcal{S}(\langle\rangle) = NONCE_H$ where $NONCE_H$ is the type consisting of all 16-byte numbers;
2. $\mathcal{S}(\langle x_1\rangle) = \{\text{kd}\}$;
3. $\mathcal{S}(\langle x_1, x_2\rangle) = MESSAGE\_LENGTH$, where $MESSAGE\_LENGTH$ is the type consisting of all 4-byte numbers;
4. $\mathcal{S}(\langle x_1, x_2, x_3\rangle) = \{\text{sig}\}$;
5. $\mathcal{S}(\langle x_1, x_2, x_3, x_4\rangle) = KDLENGTH$, where $KDLENGTH$ is the type consisting of all 2-byte numbers;
6. $\mathcal{S}(\langle x_1, x_2, x_3, x_4, x_5\rangle) = KDHEADER$, where the type $KDHEADER$ consists of all possible KD headers whose length is less than $x_3 - \text{length}(x_1||x_2||x_3||x_4||x_5)$ and the value of x_5.
7. $\mathcal{S}(\langle x_1, x_2, x_3, x_4, x_5, x_6\rangle) = KEYS$, where $KEYS$ is the set of all numbers whose length is less than $x_3 - \text{length}(x_1||x_2||x_3||x_4||x_5||x_6)$ and equal to $x_5 - \text{length}(x_6)$. Note that the second constraint makes the first redundant.

All of the above types are local to the receiver, but under the control of the sender.

We begin by creating the first three child nodes. All three cases length(y_1) = length(x_1), length(y_1) < length(x_1), and length(y_1) > length(x_1), are non-trivial, since $x_1 \in NONCE_H$ is an arbitrary 16-byte number, and $y_1 \in NONCE_A$ is a completely arbitrary number. Hence the probability of $NONCE_A \sqcap NONCE_B$ is one in all cases. But let's look at the children of these nodes. For the node corresponding to length(y_1) = length(x_1), we need to compare x_2 and y_2. The term x_2 is the payload identifier corresponding to "kd". It is one byte long.

The term y_2 is the random nonce $NONCE_B$ generated by the GCKS. Since y_2 is the last field in the POP, there is only one possibility; that is, length(x_2) < length(y_2). But this would require a member of $Pre(NONCE_B)$ to be equal to "kd". Since $NONCE_B$ is local to the GCKS and under its control, the chance of this is $1/2^8$. If this is not too small to worry about, we construct the child of this node. Again, there will be only one, and it will correspond to length(x_3) < length(y_2) - length(x_2). In this case, x_3 is the apparently arbitrary number $MESSAGE\_LENGTH$. But there is a nontrivial relationship between $MESSAGE\_LENGTH$ and $NONCE_B$, in that $MESSAGE\_LENGTH$ must describe a length equal to $M + N$, where M is the length of the part of $NONCE_B$ remaining after the point at which $MESSAGE\_LENGTH$ appears in it, and N describes the length of the signature payload. Since both of these lengths are outside of the intruder's control, the probability that the first part of $NONCE_B$ will have exactly this value is $1/2^{16}$. We are now up to a probability of $1/2^{24}$.

When we go to the next child node, again the only possibility is length(x_4) < length(y_2) - length(x_3) - length(x_2), and the comparison in this case is with the 1-byte representation of "sig". The probability of type confusion now becomes $1/2^{32}$. If this is still a concern, we can continue in this fashion, comparing pieces of $NONCE_B$ with the components of the Groupkey Push Message until the risk has been reduced to an acceptable level. A similar line of reasoning works for the case length(y_1) < length(x_1).

We now look at the case length(y_1) > length(x_1), and show how it can be used to construct the attack we mentioned at the beginning of this paper. We concentrate on the child node generated by the constraint length(y_1) - length(x_1) > length(x_2). Since $y_1 \in NONCE_A$ is an arbitrary number, the probability that x_2 can be taken for a piece of y_1, given the length constraint, is one. We continue in this fashion, until we come to the node generated by the constraint length(x_7) <length(y_1) - $\sum_{i=1}^{5} x_i$. The remaining field of the Groupkey Pull Message, $x_7 \in KEYS$ is an arbitrary number, so the chance that the remaining field of the POP, y_2 together with what remains of y_1, can be mistaken for x_7, is one, since the concatenation of the remains of y_1 with y_2, by definition, will be a member of the abitrary set $KEYS$.

7 Conclusion and Discussion

We have developed a procedure for determining whether or not type confusions are possible in signed messages in a cryptographic protocol. Our approach has certain advantages over previous applications of formal methods to type confusion; we can take into account the possibility that an attacker could cause pieces of message fields to be confused with each other, as well as entire fields. It also takes into account the probability of an attack succeeding. Thus, for example, it would catch message type attacks in which typing tags, although present, are so short that it is possible to generate them randomly with a non-trivial probability.

Our greater generality comes at a cost, however. Our procedure is not guaranteed to terminate for unbounded message types, and even for bounded types it is exponential in the number of message fields. Thus, it would have not have terminated for the actual, unsimplified, GDOI protocol, which allows an arbitrary number of keys in the Key Download payload, although it still would have found the type confusion attacks that we described at the beginning of this paper.

Also, we have left open the problem of how the probabilities are actually computed, although in many cases, such as that of determining whether or not a random number can be mistaken for a formatted field, this is fairly straightforward. In other cases, as in the comparison between $NONCE_B$ and $MESSAGE\_LENGTH$ from above, things may be more tricky. This is because, even though the type of a field is a function of the fields that come before it in a message, the values of the fields that come after it may also act as a constraint, as the length of the part of the message appearing after $MESSAGE\_LENGTH$ does on the value of $MESSAGE\_LENGTH$.

Other subtleties may arise from the fact that other information that may or may not be available to the intruder may affect the probability of type confusion. For example, in the comparison between $MESSAGE\_LENGTH$ and $NONCE_B$, the intruder has to generate $NONCE_A$ before it sees $NONCE_B$. If it could generate $NONCE_A$ after it saw $NONCE_B$, this would give it some more control over the placement of $MESSAGE\_LENGTH$ with respectt to $NONCE_B$. This would increase the likelyhood that it would be able to force $MESSAGE\_LENGTH$ to have the appropriate value.

But, although we will need to deal with special cases like these, we believe that, in practice, the number of different types of such special cases will be small, and thus we believe that it should be possible to narrow the problem down so that a more efficient and easily automatable approach becomes possible. In particular, a study of the most popular approaches to formatting cryptographic protocols should yield some insights here.

8 Acknowledgements

We are greatful to MSec and SMuG Working Groups, and in particular to the authors for the GDOI protocol, for many helpful discussions on this topic. This work was supported by ONR.

Bibliography

[1] J. Alves-Foss. Provably insecure mutual authentication protocols: The two party symmetric encryption case. In *Proc. 22nd National Information Systems Security Conference.*, Arlington, VA, 1999.

[2] Mark Baugher, Thomas Hardjono, Hugh Harney, and Brian Weis. The Group Domain of Interpretation. Internet Draft draft-ietf-msec-gdoi-04.txt, Internet Engineering Task Force, February 26 2002. available at http://www.ietf.org/internet-drafts/draft-ietf-msec-gdoi-04.txt.

[3] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, Internet Engineering Task FOrce, November 1998. available at http://ietf.org/rfc/rfc2409.txt.

[4] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 255–268. IEEE Computer Society Press, June 2000. A revised version is to appear in the *Journal of Computer Security*.

[5] John Kelsey and Bruce Schneier. Chosen interactions and the chosen protocol attack. In *Security Protocols, 5th International Workshop April 1997 Proceedings*, pages 91–104. Springer-Verlag, 1998.

[6] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Request for Comments 2408, Network Working Group, November 1998. Available at http://ietf.org/rfc/rfc2408.txt.

[7] Catherine Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In *Proceedings of ESORICS '96*. Springer-Verlag, 1996.

[8] Einar Snekkenes. Roles in cryptographic protocols. In *Proceedings of the 1992 IEEE Computer Security Symposium on Research in Security and Privacy*, pages 105–119. IEEE Computer Society Press, May 4-6 1992.

Proving Cryptographic Protocols Safe From Guessing Attacks

Ernie Cohen

Microsoft Research, Cambridge UK
ernie.cohen@acm.org

Abstract

We extend the first-order protocol verification method of [1] to prove crypto protocols secure against an active adversary who can also engage in idealized offline guessing attacks. The enabling trick is to automatically construct a first-order structure that bounds the deduction steps that can appear in a guessing attack, and to use this structure to prove that such attacks preserve the secrecy invariant. We have implemented the method as an extension to the protocol verifier TAPS, producing the first mechanical proofs of security against guessing attacks in an unbounded model.

1 Introduction

Many systems implement security through a combination of strong secrets (e.g., randomly generated keys and nonces) and weak secrets (e.g. passwords and PINs). For example, many web servers authenticate users by sending passwords across SSL channels. Although most formal protocol analysis methods treat strong and weak secrets identically, it is well known that special precautions have to be taken to protect weak secrets from offline guessing attacks.

For example, consider the following simple protocol, designed to deliver an authenticated message T from a user A to a server S:

$$A \rightarrow S : \{Na, T\}_{k(A)}$$

(Here $k(A)$ is a symmetric key shared between A and S, and Na is a freshly generated random nonce.) If $k(A)$ is generated from a weak secret, an adversary might try to attack this protocol as follows:

- If A can be tricked into sending a T that is in an easily recognized sparse set (e.g., an English text, or a previously published message), the adversary can try to guess A's key. He can then confirm his guess by decrypting the message and checking that the second component is in the sparse set.
- If A can be tricked into sending the same T twice (using different nonces), or if another user B can be tricked into sending the same T, the adversary can try to guess a key for each message. He can then confirm his guess by decrypting the messages with the guessed keys, and checking that their second components are equal.

In these attacks, the adversary starts with messages he has seen and computes new messages with a sequence of *steps* (guessing, decryption, and projection in these examples). Successful attack is indicated by the discovery of an unlikely coincidence — a message (T in the examples above) that either has an unlikely property (as in the first example) or is generated in two essentially different ways (as in second example). Such a message is called a *verifier* for the attack; intuitively, a verifier confirms the likely correctness of the guesses on which its value depends.

This approach to modelling guessing attacks as a search for coincidence was proposed by Gong et. al. [2], who considered only the first kind of verifier. Lowe [3] proposed a model that includes both kinds of verifiers [1]. He also observed that to avoid false attacks, one should ignore verifiers produced by steps that simply reverse other derivation steps. For example, if an adversary guesses a symmetric key, it's not a coincidence that encrypting and then decrypting a message with this key yields the original message (it's an algebraic identity); in Lowe's formulation, such a coincidence is ignored because the decryption step "undoes" the preceding encryption step.

Lowe also described an extension to Casper/FDR that searches for such attacks by looking for a trace consisting of an ordinary protocol execution (with an active adversary), followed by a sequence of steps (dependent on a guess) that leads to a verifier. However, he did not address the problem of proving protocols safe from such attacks.

In this paper we extend the first-order verification method of [1] to prove protocols secure in the presence of an active attacker that can also engage in these kinds of offline guessing attacks. We have also extended our verifier, TAPS, to construct these proofs automatically; we believe them to be the first mechanical proofs of security against a guessing attacker in an unbounded model.

[1] Lowe also considered the possibility of using a guess as a verifier; we handle this case by generating guesses with explicit attacker steps.

1.1 Roadmap

Our protocol model and verification method are fully described in [1]; in this paper, we provide only those details needed to understand the handling of guessing attacks.

We model protocols as transition systems. The state of the system is given by the set of events that have taken place (e.g., which protocol steps have been executed by which principals with which message values). We model communication by keeping track of which messages have been *published* (i.e., sent in the clear); a message is sent by publishing it, and received by checking that it's been published. The adversary is modelled by actions that combine published messages with the usual operations ((un)pairing, (en/de)cryption, etc.) and publish the result.

Similarly, a guessing attacker uses published messages to construct a guessing attack (essentially, a sequence of steps where each step contributes to a coincidence) and publishes any messages that appear in the attack (in particular, any guesses). We describe these attacks formally in section 2. Our model differs from Lowe's in minor ways (it's a little more general, and we don't allow the attack to contain redundant steps); however, we prove in the appendix that for a guesser with standard attacker capabilities, and with a minor fix to his model, the models are essentially equivalent.

To verify a protocol, we try to generate an appropriate set of first-order invariants, and prove safety properties from the invariants by first-order reasoning. Most of these invariants are invariant by construction; the sole exception is the *secrecy invariant* (described in section 4), which describes conditions necessary for the publication of a message. [1] describes how TAPS constructs and checks these invariants; in this paper, we are concerned only with how to prove that guessing attacks maintain the secrecy invariant.

In section 3, we show how to prove a bound on the information an adversary learns from a guessing attack by constructing a set of steps called a *saturation*; the main theorem says that if a set of messages has a saturation that has no verifiers, then a guessing attack starting with information from that set cannot yield information outside that set. In section 5, we show how to automatically generate first-order saturations suitable to show preservation of TAPS secrecy invariants. TAPS uses a resolution theorem prover to check that the saturation it constructs really does define a saturation, and to check that this saturation has no verifiers.

2 Guessing Attacks

In this section, we describe guessing attacks abstractly. We also define a *standard model*, where the attacker has the usual Dolev-Yao adversary capabilities (along with the ability to guess and recognize).

We assume an underlying set of *messages*; variables X, Y, Z, A, Na, and T range over messages, and M over arbitrary sets of messages. In the standard model, the message space comes equipped with the following functions (each injective[2], with disjoint ranges):

nil	a trivial message
$\{X, Y\}$	the ordered pair formed from X and Y
X_Y	encryption of X under the key Y

Following Lowe, we define a guessing attacker by the *steps* that he can use to construct an attack; variables starting with s denote steps, variables starting with S denote sets of steps or finite sequences of steps without repetition (treating such sequences as totally ordered sets). We assume that each step has a finite sequence of message *inputs* and a single message *output*; intuitively, a step models an operation available to the adversary that, produces the output from the inputs. An input/output of a set of steps is an input/output of any of its members. We say s is *inside* M iff the inputs and output of s are all in M, and is *outside* M otherwise. S is inside M iff all its steps are inside M.

In the standard model, a step is given by a vector of length three, the first and third elements giving its inputs and output, respectively. Steps are of the following forms, where d,$guessable$, and $checkable$ are protocol-dependent predicates described below:

$\langle\langle\rangle, nil, nil\rangle$
$\langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
$\langle\langle X, Y\rangle, enc, X_Y\rangle$
$\langle\langle\{X, Y\}\rangle, car, X\rangle$
$\langle\langle\{X, Y\}\rangle, cdr, Y\rangle$
$\langle\langle X_Y, Z\rangle, dec, X\rangle$, where $d(Y, Z)$
$\langle\langle\rangle, guess, X\rangle$ where $guessable(X)$
$\langle\langle X\rangle, check, nil\rangle$ where $checkable(X)$

The first five steps model the adversary's ability to produce the empty message, to pair, encrypt and project. The sixth step models his ability to decrypt; $d(X, Y)$ means that messages encrypted under X can be decrypted using Y[3]. The seventh step models the adversary's ability to guess; we think of the adversary as having a long list of likely secrets, and $guessable(X)$ just means that X is in the list. The last step models the adversary's ability to recognize members of particular sparse sets (such as English texts); $checkable(X)$ means X passes the recognition test[4].

[2] As usual, we cheat in assuming pairing to be injective; to avoid a guesser being able to recognize pairs, pairing is normally implemented as concatenation, which is not generally injective. One way to improve the treatment would be to make message lengths explicit.

[3] For example, the axiom $sk(X) \Leftrightarrow (\forall Y : d(X, Y) \Leftrightarrow X = Y)$ defines sk as a recognizer for symmetric keys.

[4] Because of the way we define attacks, $checkable$ has to be defined with some care. For example, defining $checkable$ to be all messages of the form $\{X, X\}$ would allow construction of a verifier for any guess. However, we can use $checkable$ to recognize asymmetric key pairs, which were treated by Lowe as a special case.

Let *undoes* be a binary relation on steps, such that if $s1$ undoes $s2$, then the output of $s1$ is an input of $s2$ and the output of $s2$ is an input of $s1$[5]. Intuitively, declaring that $s1$ undoes $s2$ says that performing $s1$ provides no new "information" if we've already performed $s2$[6]. Typically, data constructors and their corresponding destructors are defined to undo each other.

In the standard model, we define that step $s1$ undoes step $s2$ iff there are X, Y, Z such that one of the following cases holds:

1. $s1 = \langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
   $s2 = \langle\langle\{X, Y\}\rangle, car, X\rangle$
2. $s1 = \langle\langle X, Y\rangle, cons, \{X, Y\}\rangle$
   $s2 = \langle\langle\{X, Y\}\rangle, cdr, Y\rangle$
3. $s1 = \langle\langle X, Y\rangle, enc, X_Y\rangle$
   $s2 = \langle\langle X_Y, Z\rangle, dec, X\rangle$
4. any of the previous cases with $s1$ and $s2$ reversed

A sequence of steps S is a *pre-attack* on M if (1) every input to every step of S is either in M or the output of an earlier step of S, and (2) no step of S undoes an earlier step of S. An *M-verifier* of a set of steps S is the output of a step of S that is in M or the output of another step of S. A pre-attack on M, S, is an *attack* on M iff the output of every step of S is either an M-verifier of S or the input of a later step of S. Intuitively, an attack on M represents an adversary execution, starting from M, such that every step generates new information, and every step contributes to a coincidence (i.e., a path to a verifier). Note that attacks are monotonic: if $M \subseteq M'$, an attack on M is also an attack on M'. An attack on M is *effective* iff it contains a step outside M. Note also that removing a step inside M from an (effective) attack on M leaves an (effective) attack on M.

A guessing attacker is one who, in addition to any other capabilities, can publish any messages that appear as inputs outputs in an attack on the set of published messages[7].

example: We illustrate with some examples taken from [3]. In each case, we assume g is guessable.

- If $g_g \in M$, then
  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g, g\rangle, enc, g_g\rangle$$
  is an attack on M
- If $g_g \in M$ and g is a symmetric key, then
  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g_g, g\rangle, dec, g\rangle$$
  is an attack on M
- If $M = \emptyset$ and g is a symmetric key, then
  $$\langle\langle\rangle, guess, g\rangle, \langle\langle g, g\rangle, enc, g_g\rangle, \langle\langle g_g, g\rangle, dec, g\rangle$$
  is not an attack, or even a pre-attack, on M, because the last step undoes the second step.
- If $\{v, v\}_g \in M$, where g is a symmetric key,
  $$\langle\langle\rangle, guess, g\rangle, \langle\langle\{v, v\}_g, g\rangle, dec, \{v, v\}\rangle,$$
  $$\langle\langle\{v, v\}\rangle, car, v\rangle, \langle\langle\{v, v\}\rangle, cdr, v\rangle$$
  is an attack on M.

If M in the last example is the set of published messages, the attack has the effect of publishing the messages $g, \{v, v\}_g, \{v, v\}$, and v.
end of example

3 Saturation

How can we prove that there are no effective attacks on a set of messages M? One way is to collect together all messages that might be part of an effective attack on M, and show that this set has no verifiers. However, this set is inductively defined and therefore hard to reason about automatically. We would rather work with a first-order approximation to this set (called a saturation below); this approximation might have additional steps (e.g., a set of steps forming a loop, or steps with an infinite chain of predecessors); nevertheless, as long as it has no verifiers, it will suit our purposes.

However, there is a catch; to avoid false attacks, we don't want to add a step to the set if it undoes a step already in the set. Thus, adding extra steps to the set might cause us to not include other steps that should be in the set (and might be part of an effective guessing attack). Fortunately, it turns out that this can happen only if the set has a verifier.

Formally, a set of steps S *saturates* M iff every step outside M whose inputs are all in $M \cup outputs(S)$ is either in S or undoes some step in S. The following theorem says that we can prove M free from effective attacks by constructing a saturation of M without an M-verifier, and that this method is complete:

[5]The peculiar requirement on undoing has no real intuitive significance, but is chosen to make the proof of the saturation theorem work. The requirement that the output of $s1$ is an input of $s2$ is not strictly necessary, but it makes the proof of completeness part of the theorem easier.

[6]One of the weaknesses of our model (and Lowe's) is that the *undoes* relation does not distinguish between it's inputs. For example, encryption with a public key followed by decryption with a private key does not reveal any new information about the plaintext, but does reveal information about the keys (namely, that they form a key pair). This is why asymmetric key pairs have to be handled with *check* steps.

[7]Because the set of published messages is generally not closed under adversary actions, an attack might publish new messages even without using guesses. However, such any message published through such an attack could equally be published through ordinary adversary actions.

Theorem 1 *There is an effective attack on M iff every saturation of M has an M-verifier.*

To prove the forward implication, let A be an effective attack on M, and let S saturate M; we show that S has an M-verifier. Since A is effective, assume wlog that A has no steps inside M. If $A \subseteq S$, the M-verifier for A is also an M-verifier for S. Otherwise, let a be the first step of A not in S; all inputs of a are in outputs of previous steps of A (which are also steps of S) or in M. Since S saturates M and $a \notin S$, a undoes some step $s \in S$; by the condition on undoing, the output x of s is an input of a. Moreover, because A is a pre-attack, x is either in M or the output of an earlier step $a1$ of A (hence $a1 \in S$) that a does not undo (and hence, since a undoes s, $a1 \neq s$). In either case, x is an M-verifier for S.

To prove the converse, let S be the set of all steps of all pre-attacks on M whose steps are outside M; we show that (1) S saturates M and (2) if S has a verifier, there is an effective attack on M. To show (1), let s be a step outside M whose inputs are all in $S \cup M$ and does not undo a step of S; we have to show that $s \in S$. Since each input of s that is not in M is the output of a step of a pre-attack on M, concatenate these pre-attacks together (since s has only finitely many inputs) and repeatedly delete steps that undo earlier steps (by the condition on the $undoes$ relation, such deletions do not change the set of messages that appear in the sequence); the result is a pre-attack on M. Since s does not undo any step of S, it does not undo any steps of this sequence, so adding it to the end gives a pre-attack on M; thus $s \in S$. To show (2), suppose S has an M-verifier x; then there are steps $s1, s2 \in S$ such that x is the output of $s1$ and $s2$ and either $x \in M$ or $s1 \neq s2$. By the definition of S, $s1$ and $s2$ are elements of pre-attacks A and B with steps outside M, and x is an M-verifier for $A \cup B$. Thus, $A; B$ is a pre-attack on M with a verifier; by repeatedly removing from this pre-attack steps that produce outputs that are neither M-verifiers nor inputs to later steps, we are eventually left with an effective attack on M.

4 The secrecy invariant

In our verification method (for nonguessing attackers), the secrecy invariant has the form

$$pub(X) \Rightarrow ok(X)$$

where $pub(X)$ is a state predicate that means X has been published and ok is the strongest predicate[8] satisfying the following formulas:

$$\begin{array}{rcl} prime(X) & \Rightarrow & ok(X) \\ & & ok(nil) \\ pub(X) \wedge pub(Y) & \Rightarrow & ok(\{X, Y\}) \\ pub(X) \wedge pub(Y) & \Rightarrow & ok(X_Y) \end{array}$$

The computation of an appropriate definition of the predicate $prime$ and the proof obligations needed to show the invariance of the secrecy invariant are fully described in [1]; for our purposes, the relevant properties of $prime$ are

$$\begin{array}{rcl} prime(\{X, Y\}) & \Rightarrow & prime(X) \wedge prime(Y) \\ prime(X_Y) \wedge dk(Y) & \Rightarrow & prime(X) \end{array}$$

where $dk(X) \Leftrightarrow (\exists\, Y : d(X, Y) \wedge pub(Y))$. Intuitively, $dk(X)$ means that the adversary possesses a key to decrypt messages encrypted under X.

example (continued): Consider the protocol described in the introduction. To simplify the example, we assume that $k(A)$ is unpublished, for all A, and that Na and T are freshly generated every time the protocol is run. For this protocol, TAPS defines $prime$ to be the strongest predicate satisfying

$$p0(A, Na, T) \quad \Rightarrow \quad prime(\{Na, T\}_{k(A)}) \qquad (1)$$

where $p0(A, Na, T)$ is a state predicate that records that A has executed the first step of the protocol with nonce Na and message T. (Note that if some keys could be published, the definition would include additional primality cases for $\{Na, T\}$, Na, and T.)
end of example

To prove that the secrecy invariant is preserved by guessing steps, we want to prove that any value published by a guessing attack is ok when the attacker publishes it. Inductively, we can assume that the secrecy invariant holds when a guessing attack is launched, so we can assume that every published message is ok. Because attacks are monotonic in the starting set of messages, it suffices to show that there are no effective attacks on the set of ok messages; by the saturation theorem, we can do this by constructing a saturation without verifiers for the set of ok messages.

5 Generating a Saturation

We define our saturation as follows; in each case, we eliminate steps inside ok to keep the saturation as small as possible. We say a message is *available* if it is ok or is the output of a step of the saturation. We call car,cdr, and dec steps *destructor* steps, and the remaining steps *constructor* steps. The *main input* of a destructor step is the encrypted message in the case of a dec step and the sole input in the case of a car or cdr step. A constructor step outside ok is

[8] Because ok does not occur in the definition of $prime$, we could instead define ok as an ordinary disjunction, but it's more convenient to work with sets of formulas than big disjunctions. The same remarks apply to the predicates $guessable$, $checkable$, cc, scc, sd, and $coreOK$ below.

in the saturation iff its inputs are available and it does not undo a destructor step of the saturation. A destructor step outside ok is in the saturation if it is a *core* step (defined below); we define the core so that it includes any destructor step outside ok whose inputs are available and whose main input is either $prime$, the output of a guessing step, or the output of a core step.

To see that this definition is consistent, note that if we take the same definition but include in the core all constructor steps outside ok whose inputs are available (i.e. ignoring whether they undo destructor steps), the conditions on the saturation and availability form a Horn theory, and so have a strongest solution. Removing from the saturation those constructor steps that undo core steps doesn't change the definition of availability or the core (since the main input to every core step is either $prime$ or the output of a core step).

To show that this definition yields a saturation of ok, we have to show that any step outside ok whose inputs are available is either in the saturation or undoes a step of the saturation. This is immediate for constructor steps and for destructor steps whose main input is $prime$, a guess, or the output of a core step. A destructor step whose main input is the output of a constructor step in the saturation (other than a $guess$) undoes the constructor step (because encryption and pairing are injective with disjoint ranges). Finally, for a destructor step whose main input is ok but not $prime$, the main input must be a pair or encryption whose arguments are published (and hence ok by the secrecy invariant), so such a destructor step is inside ok.

Finally, we need to define the core. Let $sd(X_Y)$ mean that $\langle\{X_Y, Z\}, dec, X\rangle$ is a core step for some Z[9], let $scc(\{X,Y\})$ mean that $\langle\langle\{X,Y\}\rangle, car, X\rangle$ and $\langle\langle\{X,Y\}\rangle, cdr, Y\rangle$ are core steps[10], and let $cc(X)$ ("X is a core candidate") mean that X is either prime, a guess, or the output of a core step. For the remainder of the description, we assume that guessability and primality are given as the strongest predicates satisfying a finite set of equations of the forms $\mathsf{f} \Rightarrow prime(\mathsf{X})$ and $\mathsf{f} \Rightarrow guessable(\mathsf{X})$. Let $\mathcal{F}$ be the set of formulas $\mathsf{f} \Rightarrow cc(\mathsf{X})$, where $\mathsf{f} \Rightarrow prime(\mathsf{X})$ is a formula defining $prime$ or $\mathsf{f} \Rightarrow guessable(\mathsf{X})$ is a formula defining $guessable$. These formulas guarantee that *cc* includes all prime messages and all guessable messages.

example (continued): For our example, we assume that all keys are guessable, so g is defined as the strongest predicate satisfying

$$guessable(k(A)) \tag{2}$$

[9]We don't need to keep track of what decryption key was used (when creating the saturation), because the $undoes$ relation for decryptions is independent of decryption key. However, when we check for verifiers, we have to make sure that at most one decryption key is available for each of these encryptions.

[10]For the saturation defined here, each of these steps are in the core iff $cc(\{X,Y\})$

Since $prime$ and $guessable$ are defined by formulas (1) and (2) respectively, $\mathcal{F}$ initially contains the formulas

$$p0(A, Na, T) \Rightarrow cc(\{Na, T\}_{k(A)}) \tag{3}$$
$$cc(k(A)) \tag{4}$$

end of example

To make sure that $\mathcal{F}$ defines a suitable core (as defined above), we expand $\mathcal{F}$ to additionally satisfy the formulas

$$sd(X_Y) \Rightarrow cc(X) \tag{5}$$
$$scc(\{X,Y\}) \Rightarrow cc(X) \tag{6}$$
$$scc(\{X,Y\}) \Rightarrow cc(Y) \tag{7}$$
$$cc(\{X,Y\}) \wedge sc(\{X,Y\}) \Rightarrow scc(\{X,Y\}) \tag{8}$$
$$cc(X_Y) \wedge se(X_Y) \Rightarrow sd(X_Y) \tag{9}$$

where

$$se(X_Y) \Leftrightarrow (\exists\, Z : d(Y,Z) \wedge avail(Z) \wedge (\neg ok(X_Y) \vee \neg ok(Z) \vee \neg ok(X)))$$
$$sc(\{X,Y\}) \Leftrightarrow \neg ok(\{X,Y\})$$

(5)-(7) say that cc includes all outputs of core steps; (8) says that a car or cdr step is in the core if it is outside ok and its input is in cc; and (9) says that a decryption step is in the core if it is outside ok and a decryption key is available. To construct the formulas for sd, scc, and cc meeting these conditions, we repeatedly add formulas to $\mathcal{F}$ as follows:

- If $(\mathsf{f} \Rightarrow cc(\mathsf{X_Y})) \in \mathcal{F}$, add to $\mathcal{F}$ the formulas

$$\mathsf{f} \wedge se(\mathsf{X_Y}) \Rightarrow sd(\mathsf{X_Y})$$
$$\mathsf{f} \wedge se(\mathsf{X_Y}) \Rightarrow cc(\mathsf{X})$$

example (continued): Applying this rule to the formula (3), we add to $\mathcal{F}$ the formulas

$$p0(A, Na, T) \wedge se(\{Na, T\}_{k(A)}) \Rightarrow sd(\{Na, T\}_{k(A)})$$
$$p0(A, Na, T) \wedge se(\{Na, T\}_{k(A)}) \Rightarrow cc(\{Na, T\})$$

Since $k(A)$ is a symmetric key, $d(k(A), V)$ simplifies to $V = k(A)$, and both $avail(k(A))$ and $se(\{Na, T\}_{k(A)})$ simplify to $true$, so we can simplify these formulas to

$$p0(A, Na, T) \Rightarrow sd(\{Na, T\}_{k(A)}) \tag{10}$$
$$p0(A, Na, T) \Rightarrow cc(\{Na, T\}) \tag{11}$$

end of example

- If $(\mathsf{f} \Rightarrow cc(\{\mathsf{X},\mathsf{Y}\})) \in \mathcal{F}$, add to $\mathcal{F}$ the formulas

$$\begin{array}{lcl} \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & scc(\{\mathsf{X},\mathsf{Y}\}) \\ \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & cc(\mathsf{X}) \\ \mathsf{f} \wedge sc(\{\mathsf{X},\mathsf{Y}\}) & \Rightarrow & cc(\mathsf{Y}) \end{array}$$

example (continued): Applying this rule to (11), and using the fact that with the given secrecy invariant, $p0(A,Na,T) \Rightarrow \neg ok(\{Na,T\})$, yields the formulas

$$\begin{array}{lcll} p0(A,Na,T) & \Rightarrow & scc(\{Na,T\}) & (12) \\ p0(A,Na,T) & \Rightarrow & cc(Na) & (13) \\ p0(A,Na,T) & \Rightarrow & cc(T) & (14) \end{array}$$

end of example

- If $(\mathsf{f} \Rightarrow cc(\mathsf{X})) \in \mathcal{F}$, where X is a variable symbol or an application of a function other than $cons$ or enc, add to $\mathcal{F}$ the formula

$$\mathsf{f} \Rightarrow coreOK(\mathsf{X})$$

where $coreOK$ is the strongest predicate such that

$$\begin{array}{lcl} coreOK(X) & \Leftrightarrow & (sc(X) \Rightarrow scc(X)) \\ & & \wedge (se(X) \Rightarrow sd(X)) \end{array}$$

Intuitively, $coreOK(X)$ means that we believe that we don't need to generate any more core cases to handle X, because we think that f implies that if X is the main input of a destructor step, then it is already handled by one of the other cases in $\mathcal{F}$. (Typically, it is because we think that the authentication properties of the protocol guarantee that X will not be an encryption or pair[11].) This bold assumption is just a good heuristic[12]; if it turns out to be wrong, TAPS is unable to prove the resulting proof obligation, and the whole protocol proof fails (i.e., the assumption does compromise soundness).

example (continued): Applying this rule to (13),(14), and (4) yields the obligations

$$\begin{array}{lcll} p0(A,Na,T) & \Rightarrow & coreOK(Na) & (15) \\ p0(A,Na,T) & \Rightarrow & coreOK(T) & (16) \\ & & coreOK(k(A)) & (17) \end{array}$$

end of example

After this process reaches a fixed point[13], we define sd to be the strongest predicate satisfying the formulas of $\mathcal{F}$ of the form $\mathsf{f} \Rightarrow sd(\mathsf{X})$, and similarly for scc and cc.

example (continued): The formulas of $\mathcal{F}$ generate the definitions

$$\begin{array}{rcl} sd(\{Na,T\}_{k(A)}) & \Leftrightarrow & p0(A,Na,T) \\ scc(\{Na,T\}) & \Leftrightarrow & p0(A,Na,T) \end{array}$$

end of example

These definitions guarantee that the core satisfies (5)-(7); to show that it satisfies (8)-(9), we also need to prove explicitly the remaining formulas from $\mathcal{F}$ of the form $\mathsf{f} \Rightarrow coreOK(X)$. TAPS delegates these proofs to a resolution prover; if these proofs succeed, we have successfully defined the core, and hence, the saturation[14].

example (continued): The remaining obligations are (15)-(16) and (17). (15)-(16) follow trivially from the fact that $p0(A,Na,T)$ implies that A,Na, and T are all atoms; (17) depends on a suitable axiom defining the keying function k (e.g., that $k(A)$ is an atom, or is an encryption under a hash function).
end of example

Finally, we have to prove that the saturation has no verifiers. Because of the way in which our saturation is constructed, we can eliminate most of the cases as follows. The output of a $cons$ step cannot be the output of any other step of the saturation —

- it can't be the output of a core step (because the core would include the car and cdr steps generated from the output, which would be undone by the $cons$ step);
- it can't be the output of another $cons$ step (because pairing is injective) or an enc step (since $cons$ and enc have disjoint ranges);
- it can't be ok, because both of its inputs would also be published (by the definition of ok), hence ok, and so the step would be inside ok (hence out of the saturation).

Similarly, the output of an enc step cannot be the output of another enc step (because enc is injective).

Thus, we can prove the saturation has no verifiers by showing that each of the remaining cases is impossible:

[11] Unfortunately, a proof that shows this interplay between authentication reasoning a safety from guessing would be too large to present here; analogous proofs that show the synergy between authentication and secrecy reasoning are given in [1].

[12] Like all such choices in TAPS, the user can override this choice with hints about how to construct the saturation.

[13] The process reaches a fixed point by induction on the size of the term on the right of the implication; it does not require an induction principle on messages.

[14] In fact, the way the TAPS secrecy invariant is constructed, these proofs are probably not necessary. But they also present no problems if TAPS can prove the secrecy invariant (we have never seen one fail), so we have left them in for the sake of safety.

- an enc step whose output v is $prime$ or the output of a core step, such that no decryption key for v is available (This case is trivial for symmetric encryptions, since the key (and hence also a decryption key) is available.)
- two distinct core steps with the same output
- a core step whose output is ok or guessable
- a saturation step whose output is checkable

These cases are relatively easy to check, because they involve messages of known structure and pedigree.

example (continued): To prove that this saturation has no verifiers, we have to check the following cases:

- the output of a enc step that is $prime$ or the output of a core step, without an available decryption key: this case goes away because the only such encryptions are symmetric.
- a core step that produces an ok value; this case checks because the secrecy invariant guarantees that neither Na nor T is ok.
- a core step that produces a guessable value; this depends on an additional assumptions about k (e.g., $k(A)$ is an atom, but not a value generated for Na or T[15]).
- two distinct core steps that produce the same value. A dec step cannot collide with a car or cdr step - the former produces $cons$ terms, while the latter produce atoms. Two core dec steps cannot collide because Na is freshly chosen and thus uniquely determines A and T (and thus, by the injectivity properties of $cons$, $\{Na, T\}$ uniquely determines both $\{Na, T\}_{k(A)}$ and $k(A)$; because $k(A)$ is symmetric, it also uniquely determines the decryption key). Two car (respectively, cdr) steps cannot collide because Na (respectively, T) is freshly chosen, and hence functionally determines $\{Na, T\}$. Finally, a car and a cdr step cannot collide because the same value is not chosen as both an Na value and an T value. Note that these cases would fail if the same value could be used for Na and T, or if an Na or T value could be reused.
- a saturation step whose output is checkable; in this example, this is trivial, since no messages are regarded as checkable. Were we not able to prove directly that T is uncheckable, the proof would fail.

end of example

[15] In practice, we avoid having to write silly axioms like this by generating keys like $k(A)$ with protocol actions; nonce unicity lemmas [1] then give us all these conditions for free.

In general, the total number of proof obligations grows roughly as the square of the number of core cases, in contrast to the ordinary protocol reasoning performed by TAPS, where the number of obligations is roughly linear in the number of prime cases. Moreover, many of the proofs depend on authentication properties[16] Thus, as expected, the guessing part of the verification is typically much more work than the proof of the secrecy invariant itself. For example, for EKE, TAPS proves the secrecy invariant and the authentication properties of the protocol in about half a second, while checking for guessing attacks takes about 15 seconds[17]. However, the proofs are completely automatic.

While we have applied the guessing attack extension to TAPS to verify a number of toy protocols (and a few less trivial protocols, like EKE), a severe limitation is that TAPS (like other existing unbounded crypto verifiers) cannot handle Vernam encryption. The next major revision of TAPS will remedy this situation, and allow us to more reasonably judge the usefulness of our approach to guessing.

6 Acknowledgements

This work was done while visiting Microsoft Research, Cambridge; we thank Roger Needham and Larry Paulson for providing the opportunity. We also thank Gavin Lowe for patiently fielding many stupid questions about guessing attacks, and the referees for their insightful comments.

Bibliography

[1] E. Cohen, *First-Order Verification of Cryptographic Protocols.* In *JCS* (to appear). A preliminary version appears in *CSFW* (2000)

[2] L. Gong and T. Mark and A. Lomas and R. Needham and J. Saltzer, *Protecting Poorly Chosen Secrets from Guessing Attacks.* IEEE Journal on Selected Areas in Communications, 11(5):648-656 (1993)

[3] G. Lowe, *Analyzing Protocols Subject to Guessing Attacks.* In *WITS* (2002)

[4] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić, *System description: SPASS version 1.0.0.* In *CADE 15*, pages 378–382 (1999).

[16] For example, if a protocol step publishes an encryption with a component received in a message, and the message is decrypted in a guessing attack, showing that the destructor step that extracts the component doesn't collide depends on knowing where the component came from.

[17] This is in part due to the immaturity of the implementation; it takes some time to find the right blend of theorem proving and preprocessing to optimize performance. For example, the current TAPS implementation effectively forces the prover into splitting the proof into cases, since the prover is otherwise reluctant to do resolutions that introduce big disjunctions.

A Lowe's Guessing Model

In this appendix, we describe Lowe's model for guessing attacks, and sketch a proof that for the standard model and with a single guess, Lowe's model and ours are essentially equivalent.

Throughout this section, we work in the standard model. Moreover, to match Lowe's model, we define the checkable messages to be all pairs of asymmetric keys, and define a single value g to be guessable. Let M be a fixed set of messages. An *execution* is a finite sequence of distinct steps such that every input to every step is either in M or the output of an earlier step of S[18]. A message m is *new* iff every execution from M that contains m as an input or output also includes a guessing step; intuitively, new messages cannot be produced from M without the use of g. A *Lowe attack* is an execution with steps $s1$ and $s2$ (not necessarily distinct), each with output v, satisfying the following properties:

- $s1$ is either a guessing step or has an input that is new (Lowe's condition (3));
- either (1) $s1 \neq s2$, (2) v is in M, or (3) v is an asymmetric key and v^{-1} is in M or the output of a step of the execution (Lowe's condition (4))
- neither $s1$ nor $s2$ undoes any step of the execution (Lowe's condition (5))

Theorem 2 *There is a Lowe attack on M iff there is an attack on M that reveals g.*

First, let S be a Lowe attack with $s1$, $s2$, and v as in the definition above; we show how to turn S into an attack on M that reveals g. Let $s3$ be a step other than $s1$ or $s2$. If the output of $s3$ is not an input of a later step, then deleting $s3$ from S leaves a Lowe attack revealing. If $s3$ undoes an earlier step of S, then (in the standard model) the output of $s3$ is an input of the earlier step, so again deleting $s3$ from S leaves a Lowe attack revealing g. Repeating these deletions until no more are possible leaves a Lowe attack revealing g where no step undoes an earlier step, and where the output of every step other than $s1$ and $s2$ is an input to a later step.

If $s1 \neq s2$, or if $s1 = s2$ and the output of $s1$ is in M or the output of another step of S, then the outputs of $s1$ and $s2$ are verifiers and S is an attack. Otherwise, $s1 = s2$, v is an asymmetric key, and v^{-1} is in M or the output of a step of S. If v is an input to a step that follows $s1$, then again S is an attack. If $\{v, v^{-1}\}$ is neither in M nor the output of a step of G, then add to the end of S the steps $\langle\langle v, v^{-1}\rangle, cons, \{v, v^{-1}\}\rangle, \langle\langle\{v, v^{-1}\}\rangle, check, nil\rangle$ to create an attack. Otherwise, let $s3 = \langle\langle\{v, v^{-1}\}\rangle, car, v\rangle$. If $s1 = s3$, delete $s1$ from S and add to the end of S the step $\langle\{v, v^{-1}\}\rangle, check, nil\rangle$; if $s1 \neq s3$, add $s3$ to the end of S. In either case, the resulting S is an attack.

Conversely, let S be an attack on M that reveals g. Note that since S is a pre-attack, no step of S undoes a previous step of S; since the undoing relation is symmetric in the standard model, no step of S undoes any other step of S. Since g is not derivable from M, it must be the output of a guessing step of S; without loss of generality, assume it is the output of the first step.

- If g is a verifier for S, then since g is not in M, it must also be the output of another step $s1$ of S. Because S is an attack, $s1$ does not undo any earlier step of S. Thus, the prefix of S up to and including $s1$ is a Lowe attack on M.
- If g is not a verifier for S, then g must be an input to a later step of S. Because g is new, some step of S has a new input; since S is finite, there is a step s of S with a new input such that no later step has a new input. Thus, the output of s is either not new or is a verifier of S. If the output of s is not new and not a verifier of S, append to S an execution without guessing or checking steps that outputs the output of s, deleting any added steps that undo steps of S. This produces an attack with a step whose output is a verifier for S and has an input that is new. If s is not a checking step, then S is a Lowe attack. If s is a checking step, then one input to s is an asymmetric key k that is new, such that k^{-1} is either in M or the output a step that precedes s. Since k is new, it must be the output of a step $s2$ of S that is either a guessing step or has a new input; in either case, S is a Lowe attack, with $s2$ in the role of $s1$.

[18] Lowe actually did not have guessing steps, but instead simply allowed g to appear as inputs to steps of S and as a verifier. However, because guesses did not appear as explicit steps, he missed the following case: suppose g is an asymmetric key and M is the set $\langle g^{-1}\rangle$. Obviously this should be considered to be vulnerable to a guessing attack. However, there is no attack in Lowe's original model, because any *step* producing g or g^{-1} would have to undo a previous step.

Session V

Programming Language Security

More Enforceable Security Policies

Lujo Bauer, Jarred Ligatti and David Walker

Department of Computer Science
Princeton University
Princeton, NJ 08544
{lbauer|jligatti|dpw}@cs.princeton.edu

Abstract

We analyze the space of security policies that can be enforced by monitoring programs at runtime. Our program monitors are automata that examine the sequence of program actions and transform the sequence when it deviates from the specified policy. The simplest such automaton truncates the action sequence by terminating a program. Such automata are commonly known as security automata, and they enforce Schneider's EM class of security policies. We define automata with more powerful transformational abilities, including the ability to insert a sequence of actions into the event stream and to suppress actions in the event stream without terminating the program. We give a set-theoretic characterization of the policies these new automata are able to enforce and show that they are a superset of the EM policies.

1 Introduction

When designing a secure, extensible system such as an operating system that allows applications to download code into the kernel or a database that allows users to submit their own optimized queries, we must ask two important questions.

1. What sorts of security policies can and should we demand of our system?
2. What mechanisms should we implement to enforce these policies?

Neither of these questions can be answered effectively without understanding the space of enforceable security policies and the power of various enforcement mechanisms.

Recently, Schneider [Sch00] attacked this question by defining EM, a subset of safety properties [Lam85, AS87] that has a general-purpose enforcement mechanism - a *security automaton* that interposes itself between the program and the machine on which the program runs. It examines the sequence of security-relevant program actions one at a time and if the automaton recognizes an action that will violate its policy, it terminates the program. The mechanism is very general since decisions about whether or not to terminate the program can depend upon the entire history of the program execution. However, since the automaton is only able to recognize bad sequences of actions and then terminate the program, it can only enforce safety properties.

In this paper, we re-examine the question of which security policies can be enforced at runtime by monitoring program actions. Following Schneider, we use automata theory as the basis for our analysis of enforceable security policies. However, we take the novel approach that these automata are transformers on the program action stream, rather than simple recognizers. This viewpoint leads us to define two new enforcement mechanisms: an *insertion automaton* that is able to insert a sequence of actions into the program action stream, and a *suppression automaton* that suppresses certain program actions rather than terminating the program outright. When joined, the insertion automaton and suppression automaton become an *edit automaton*. We characterize the class of security policies that can be enforced by each sort of automata and provide examples of important security policies that lie in the new classes and outside the class EM.

Schneider is cognizant that the power of his automata is limited by the fact that they can only terminate programs and may not modify them. However, to the best of our knowledge, neither he nor anyone else has formally investigated the power of a broader class of runtime enforcement mechanisms that explicitly manipulate the program action stream. Erlingsson and Schneider [UES99] have implemented inline reference monitors, which allow arbitrary code to be executed in response to a violation of the security policy, and have demonstrated their effectiveness on a range of security policies of different levels of abstraction from the Software Fault Isolation policy for the Pentium IA32 architecture to the Java stack inspection policy for Sun's JVM [UES00]. Evans and Twyman [ET99] have implemented a very general enforcement mechanism for Java that allows system designers to write arbitrary code to enforce security policies.

Such mechanisms may be more powerful than those that we propose here; these mechanisms, however, have no formal semantics, and there has been no analysis of the classes of policies that they enforce. Other researchers have investigated optimization techniques for security automata [CF00, Thi01], certification of programs instrumented with security checks [Wal00] and the use of runtime monitoring and checking in distributed [SS98] and real-time systems [KVBA+99].

Overview The remainder of the paper begins with a review of Alpern and Schneider's framework for understanding the behavior of software systems [AS87, Sch00] (Section 2) and an explanation of the EM class of security policies and security automata (Section 2.3). In Section 3 we describe our new enforcement mechanisms – insertion automata, suppression automata and edit automata. For each mechanism, we analyze the class of security policies that the mechanism is able to enforce and provide practical examples of policies that fall in that class. In Section 4 we discuss some unanswered questions and our continuing research. Section 5 concludes the paper with a taxonomy of security policies.

2 Security Policies and Enforcement Mechanisms

In this section, we explain our model of software systems and how they execute, which is based on the work of Alpern and Schneider [AS87, Sch00]. We define what it means to be a security policy and give definitions for safety, liveness and EM policies. We give a new presentation of Schneider's security automata and their semantics that emphasizes our view of these machines as sequence transformers rather than property recognizers. Finally, we provide definitions of what it means for an automaton to *enforce* a property precisely and conservatively, and also what it means for one automaton to be a more effective enforcer than another automaton for a particular property.

2.1 Systems, Executions and Policies

We specify software systems at a high level of abstraction. A *system* $\mathcal{S} = (\mathcal{A}, \Sigma)$ is specified via a set of *program actions* $\mathcal{A}$ (also referred to as events or program operations) and a set of possible executions Σ. An *execution* σ is simply a finite sequence of actions $a_1, a_2, \ldots, a_n$. Previous authors have considered infinite executions as well as finite ones. We restrict ourselves to finite, but arbitrarily long executions to simplify our analysis. We use the metavariables σ and τ to range over finite sequences.

The symbol $\cdot$ denotes the empty sequence. We use the notation $\sigma[i]$ to denote the i^{th} action in the sequence (beginning the count at 0). The notation $\sigma[..i]$ denotes the subsequence of σ involving the actions $\sigma[0]$ through $\sigma[i]$, and $\sigma[i+1..]$ denotes the subsequence of σ involving all other actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences. When τ is a prefix of σ we write $\tau \prec \sigma$.

In this work, it will be important to distinguish between uniform systems and nonuniform systems. $(\mathcal{A}, \Sigma)$ is a *uniform* system if $\Sigma = \mathcal{A}^\star$ where $\mathcal{A}^\star$ is the set of all finite sequences of symbols from $\mathcal{A}$. Conversely, $(\mathcal{A}, \Sigma)$ is a *nonuniform* system if $\Sigma \subset \mathcal{A}^\star$. Uniform systems arise naturally when a program is completely unconstrained; unconstrained programs may execute operations in any order. However, an effective security system will often combine static program analysis and preprocessing with runtime security monitoring. Such is the case in Java virtual machines, for example, which combine type checking with stack inspection. Program analysis and preprocessing can give rise to nonuniform systems. In this paper, we are not concerned with how nonuniform systems may be generated, be it by model checking programs, control or dataflow analysis, program instrumentation, type checking, or proof-carrying code; we care only that they exist.

A *security policy* is a predicate P on sets of executions. A set of executions Σ satisfies a policy P if and only if $P(\Sigma)$. Most common extensional program properties fall under this definition of security policy, including the following.

- *Access Control* policies specify that no execution may operate on certain resources such as files or sockets, or invoke certain system operations.
- *Availability* policies specify that if a program acquires a resource during an execution, then it must release that resource at some (arbitrary) later point in the execution.
- *Bounded Availability* policies specify that if a program acquires a resource during an execution, then it must release that resource by some fixed point later in the execution. For example, the resource must be released in at most ten steps or after some system invariant holds. We call the condition that demands release of the resource the *bound* for the policy.
- An *Information Flow* policy concerning inputs s_1 and outputs s_2 might specify that if $s_2 = f(s_1)$ in one execution (for some function f) then there must exist another execution in which $s_2 \neq f(s_1)$.

2.2 Security Properties

Alpern and Schneider [AS87] distinguish between *properties* and more general policies as follows. A security policy P is deemed to be a (computable) *property* when the policy has the following form.

$$P(\Sigma) = \forall \sigma \in \Sigma. \hat{P}(\sigma) \qquad \text{(PROPERTY)}$$

where $\hat{P}$ is a computable predicate on $\mathcal{A}^\star$.

Hence, a property is defined exclusively in terms of individual executions. A property may not specify a relationship between possible executions of the program. Information flow, for example, which can only be specified as a condition on a set of possible executions of a program, is not a property. The other example policies provided in the previous section are all security properties.

We implicitly assume that the empty sequence is contained in any property. For all the properties we are interested in it will always be okay not to run the program in question. From a technical perspective, this decision allows us to avoid repeatedly considering the empty sequence as a special case in future definitions of enforceable properties.

Given some set of actions $\mathcal{A}$, a predicate $\hat{P}$ over $\mathcal{A}^\star$ induces the security property $P(\Sigma) = \forall\sigma \in \Sigma.\hat{P}(\sigma)$. We often use the symbol $\hat{P}$ interchangeably as a predicate over execution sequences and as the induced property. Normally, the context will make clear which meaning we intend.

Safety Properties The *safety properties* are properties that specify that "nothing bad happens." We can make this definition precise as follows. $\hat{P}$ is a safety property if and only if for all $\sigma \in \Sigma$,

$$\neg\hat{P}(\sigma) \Rightarrow \forall\sigma' \in \Sigma.(\sigma \prec \sigma' \Rightarrow \neg\hat{P}(\sigma')) \qquad \text{(SAFETY)}$$

Informally, this definition states that once a bad action has taken place (thereby excluding the execution from the property) there is no extension of that execution that can remedy the situation. For example, access-control policies are safety properties since once the restricted resource has been accessed the policy is broken. There is no way to "un-access" the resource and fix the situation afterward.

Liveness Properties A *liveness property*, in contrast to a safety property, is a property in which nothing exceptionally bad can happen in any finite amount of time. Any finite sequence of actions can always be extended so that it lies within the property. Formally, $\hat{P}$ is a liveness property if and only if,

$$\forall\sigma \in \Sigma.\exists\sigma' \in \Sigma.(\sigma \prec \sigma' \wedge \hat{P}(\sigma')) \qquad \text{(LIVENESS)}$$

Availability is a liveness property. If the program has acquired a resource, we can always extend its execution so that it releases the resource in the next step.

Other Properties Surprisingly, Alpern and Schneider [AS87] show that any property can be decomposed into the conjunction of a safety property and a liveness property. Bounded availability is a property that combines safety and liveness. For example, suppose our bounded-availability policy states that every resource that is acquired must be released and must be released at most ten steps after it is acquired. This property contains an element of safety because there is a bad thing that may occur (*e.g.*, taking 11 steps without releasing the resource). It is not purely a safety property because there are sequences that are not in the property (*e.g.*, we have taken eight steps without releasing the resource) that may be extended to sequences that are in the property (*e.g.*, we release the resource on the ninth step).

2.3 EM

Recently, Schneider [Sch00] defined a new class of security properties called EM. Informally, EM is the class of properties that can be enforced by a *monitor* that runs in parallel with a *target program*. Whenever the target program wishes to execute a security-relevant operation, the monitor first checks its policy to determine whether or not that operation is allowed. If the operation is allowed, the target program continues operation, and the monitor does not change the program's behavior in any way. If the operation is not allowed, the monitor terminates execution of the program. Schneider showed that every EM property satisfies (SAFETY) and hence EM is a subset of the safety properties. In addition, Schneider considered monitors for infinite sequences and he showed that such monitors can only enforce policies that obey the following continuity property.

$$\forall\sigma \in \Sigma.\neg\hat{P}(\sigma) \Rightarrow \exists i.\neg\hat{P}(\sigma[..i]) \qquad \text{(CONTINUITY)}$$

Continuity states that any (infinite) execution that is not in the EM policy must have some finite prefix that is also not in the policy.

Security Automata Any EM policy can be enforced by a *security automaton* A, which is a deterministic finite or infinite state machine (Q, q_0, δ) that is specified with respect to some system $(\mathcal{A}, \Sigma)$. Q specifies the possible automaton states and q_0 is the initial state. The partial function $\delta : \mathcal{A} \times Q \to Q$ specifies the transition function for the automaton.

Our presentation of the operational semantics of security automata deviates from the presentation given by Alpern and Schneider because we view these machines as *sequence transformers* rather than simple *sequence recognizers*. We specify the execution of a security automaton A on a sequence of program actions σ using a labeled operational semantics.

The basic single-step judgment has the form $(\sigma, q) \xrightarrow{\tau}_A (\sigma', q')$ where σ and q denote the input program action sequence and current automaton state; σ' and q' denote the action sequence and state after the automaton processes a single input symbol; and τ denotes the sequence of actions that the automaton allows to occur (either the first action in the input sequence or, in the case that this action is "bad," no actions at all). We may also

refer to the sequence τ as the observable actions or the automaton output. The input sequence σ is not considered observable to the outside world.

$\boxed{(\sigma, q) \xrightarrow{\tau}_A (\sigma', q')}$

$$(\sigma, q) \xrightarrow{a}_A (\sigma', q') \qquad \text{(A-STEP)}$$

if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$

$$(\sigma, q) \xrightarrow{\cdot}_A (\cdot, q) \qquad \text{(A-STOP)}$$

otherwise

We extend the single-step semantics to a multi-step semantics through the following rules.

$\boxed{(\sigma, q) \xRightarrow{\tau}_A (\sigma', q')}$

$$\frac{}{(\sigma, q) \xRightarrow{\cdot}_A (\sigma, q)} \qquad \text{(A-REFLEX)}$$

$$\frac{(\sigma, q) \xrightarrow{\tau_1}_A (\sigma'', q'') \quad (\sigma'', q'') \xRightarrow{\tau_2}_A (\sigma', q')}{(\sigma, q) \xRightarrow{\tau_1;\tau_2}_A (\sigma', q')} \qquad \text{(A-TRANS)}$$

Limitations Erlingsson and Schneider [UES99, UES00] demonstrate that security automata can enforce important access-control policies including software fault isolation and Java stack inspection. However, they cannot enforce any of our other example policies (availability, bounded availability or information flow). Schneider [Sch00] also points out that security automata cannot enforce safety properties on systems in which the automaton cannot exert sufficient controls over the system. For example, if one of the actions in the system is the passage of time, an automaton might not be able to enforce the property because it cannot terminate an action sequence effectively — an automaton cannot stop the passage of real time.

2.4 Enforceable Properties

To be able to discuss different sorts of enforcement automata formally and to analyze how they enforce different properties, we need a formal definition of what it means for an automaton to enforce a property.

We say that an automaton A *precisely enforces* a property $\hat{P}$ on the system $(\mathcal{A}, \Sigma)$ if and only if $\forall \sigma \in \Sigma$,

1. If $\hat{P}(\sigma)$ then $\forall i.(\sigma, q_0) \xRightarrow{\sigma[..i]}_A (\sigma[i+1..], q')$ and,
2. If $(\sigma, q_0) \xRightarrow{\sigma'}_A (\cdot, q')$ then $\hat{P}(\sigma')$

Informally, if the sequence belongs to the property $\hat{P}$ then the automaton should not modify it. In this case, we say the automaton *accepts* the sequence. If the input sequence is not in the property, then the automaton may (and in fact must) edit the sequence so that the output sequence satisfies the property.

Some properties are extremely difficult to enforce precisely, so, in practice, we often enforce a stronger property that implies the weaker property in which we are interested. For example, information flow is impossible to enforce precisely using run-time monitoring as it is not even a proper property. Instead of enforcing information flow, an automaton might enforce a simpler policy such as access control. Assuming access control implies the proper information-flow policy, we say that this automaton *conservatively enforces* the information-flow policy. Formally, an automaton conservatively enforces a property $\hat{P}$ if condition 2 from above holds. Condition 1 need not hold for an automaton to conservatively enforce a property. In other words, an automaton that conservatively enforces a property may occasionally edit an action sequence that actually obeys the policy, even though such editing is unnecessary (and potentially disruptive to the benign program's execution). Of course, any such edits should result in an action sequence that continues to obeys the policy. Henceforth, when we use the term *enforces* without qualification (precisely, conservatively) we mean enforces precisely.

We say that automaton A_1 enforces a property $\hat{P}$ *more precisely* or *more effectively* than another automaton A_2 when either

1. A_1 accepts more sequences than A_2, or
2. The two automata accept the same sequences, but the average edit distance[1] between inputs and outputs for A_1 is less than that for A_2.

3 Beyond EM

Given our novel view of security automata as sequence transformers, it is a short step to define new sorts of automata that have greater transformational capabilities. In this section, we describe insertion automata, suppression automata and their conjunction, edit automata. In each case, we characterize the properties they can enforce precisely.

3.1 Insertion Automata

An *insertion automaton* I is a finite or infinite state machine (Q, q_0, δ, γ) that is defined with respect to some system of executions $\mathcal{S} = (\mathcal{A}, \Sigma)$. Q is the set of all possible

[1] The *edit distance* between two sequences is the minimum number of insertions, deletions or substitutions that must be applied to either of the sequences to make them equal [Gus97].

machine states and q_0 is a distinguished starting state for the machine. The partial function $\delta : \mathcal{A} \times Q \to Q$ specifies the transition function as before. The new element is a partial function γ that specifies the insertion of a number of actions into the program's action sequence. We call this the *insertion function* and it has type $\mathcal{A} \times Q \to \vec{\mathcal{A}} \times Q$. In order to maintain the determinacy of the automaton, we require that the domain of the insertion function is disjoint from the domain of the transition function.

We specify the execution of an insertion automaton as before. The single-step relation is defined below.

$$(\sigma, q) \xrightarrow{a}_I (\sigma', q') \qquad \text{(I-STEP)}$$

if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$

$$(\sigma, q) \xrightarrow{\tau}_I (\sigma, q') \qquad \text{(I-INS)}$$

if $\sigma = a; \sigma'$
and $\gamma(a, q) = \tau, q'$

$$(\sigma, q) \xrightarrow{\cdot}_I (\cdot, q) \qquad \text{(I-STOP)}$$

otherwise

We can extend this single-step semantics to a multi-step semantics as before.

Enforceable Properties We will examine the power of insertion automata both on uniform systems and on nonuniform systems.

Theorem 3 (Uniform I-Enforcement) *If $\mathcal{S}$ is a uniform system and insertion automaton I precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (*SAFETY*).*

Proof: Proofs of the theorems in this work are contained in our Technical Report [BLW02]; we omit them here due to space constraints. ■

If we consider nonuniform systems then the insertion automaton can enforce non-safety properties. For example, consider a carefully chosen nonuniform system $\mathcal{S}'$, where the last action of every sequence is the special stop symbol, and stop appears nowhere else in $\mathcal{S}'$. By the definition of safety, we would like to enforce a property $\hat{P}$ such that $\neg\hat{P}(\tau)$ but $\hat{P}(\tau; \sigma)$. Consider processing the final symbol of τ. Assuming that the sequence τ does not end in stop (and that $\neg\hat{P}(\tau; \mathsf{stop})$), our insertion automaton has a safe course of action. After seeing τ, our automaton waits for the next symbol (which must exist, since we asserted the last symbol of τ is not stop). If the next symbol is stop, it inserts σ and stops, thereby enforcing the policy. On the other hand, if the program itself continues to produce σ, the automaton need do nothing.

It is normally a simple matter to instrument programs so that they conform to the nonuniform system discussed above. The instrumentation process would insert a stop event before the program exits. Moreover, to avoid the scenario in which a non-terminating program sits in a tight loop and never commits any further security-relevant actions, we could ensure that after some time period, the automaton receives a timeout signal which also acts as a stop event.

Bounded-availability properties, which are not EM properties, have the same form as the policy considered above, and as a result, an insertion automaton can enforce many bounded-availability properties on non-uniform systems. In general, the automaton monitors the program as it acquires and releases resources. Upon detecting the bound, the automaton inserts actions that release the resources in question. It also releases the resources in question if it detects termination via a stop event or timeout.

We characterize the properties that can be enforced by an insertion automaton as follows.

Theorem 4 (Nonuniform I-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ can be enforced by some insertion automaton if and only if there exists a function γ_p such that for all executions $\sigma \in \mathcal{A}^\star$, if $\neg\hat{P}(\sigma)$ then*

1. *$\forall\sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \neg\hat{P}(\sigma')$, or*
2. *$\sigma \notin \Sigma$ and $\hat{P}(\sigma; \gamma_p(\sigma))$*

Limitations Like the security automaton, the insertion automaton is limited by the fact that it may not be able to be able to exert sufficient controls over a system. More precisely, it may not be possible for the automaton to synthesize certain events and inject them into the action stream. For example, an automaton may not have access to a principal's private key. As a result, the automaton may have difficulty enforcing a fair exchange policy that requires two computational agents to exchange cryptographically signed documents. Upon receiving a signed document from one agent, the insertion automaton may not be able to force the other agent to sign the second document and it cannot forge the private key to perform the necessary cryptographic operations itself.

3.2 Suppression Automata

A *suppression automaton* S is a state machine (Q, q_0, δ, ω) that is defined with respect to some system of executions $\mathcal{S} = (\mathcal{A}, \Sigma)$. As before, Q is the set of all possible machine states, q_0 is a distinguished starting state for the machine and the partial function δ specifies the transition function. The partial function $\omega : \mathcal{A} \times Q \to \{-, +\}$ has the same domain as δ and indicates whether or not the action in question is to be suppressed $(-)$ or emitted $(+)$.

$$(\sigma, q) \xrightarrow{a}_S (\sigma', q') \qquad \text{(S-STEPA)}$$

if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = +$

$$(\sigma, q) \xrightarrow{\cdot}_S (\sigma', q') \qquad \text{(S-STEPS)}$$

if $\sigma = a; \sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = -$

$$(\sigma, q) \xrightarrow{\cdot}_S (\cdot, q) \qquad \text{(S-STOP)}$$

otherwise

We extend the single-step relation to a multi-step relation using the reflexivity and transitivity rules from above.

Enforceable Properties In a uniform system, suppression automata can only enforce safety properties.

Theorem 5 (Uniform S-Enforcement) *If $\mathcal{S}$ is a uniform system and suppression automaton S precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (*SAFETY*).*

In a nonuniform system, suppression automata can once again enforce non-EM properties. For example, consider the following system $\mathcal{S}$.

$$\begin{array}{rcl} \mathcal{A} & = & \{\mathsf{aq}, \mathsf{use}, \mathsf{rel}\} \\ \Sigma & = & \{\mathsf{aq}; \mathsf{rel}, \\ & & \mathsf{aq}; \mathsf{use}; \mathsf{rel}, \\ & & \mathsf{aq}; \mathsf{use}; \mathsf{use}; \mathsf{rel}\} \end{array}$$

The symbols aq, use, rel denote acquisition, use and release of a resource. The set of executions includes zero, one, or two uses of the resource. Such a scenario might arise were we to publish a policy that programs can use the resource at most two times. After publishing such a policy, we might find a bug in our implementation that makes it impossible for us to handle the load we were predicting. Naturally we would want to tighten the security policy as soon as possible, but we might not be able to change the policy we have published. Fortunately, we can use a suppression automaton to suppress extra uses and dynamically change the policy from a two-use policy to a one-use policy. Notice that an ordinary security automaton is not sufficient to make this change because it can only terminate execution.[2] After terminating a two-use application, it would be unable to insert the release necessary to satisfy the policy.

We can also compare the power of suppression automata with insertion automata. A suppression automaton cannot enforce the bounded-availability policy described in the previous section because it cannot insert release events that are necessary if the program halts prematurely. That is, although the suppression automaton could suppress all non-release actions upon reaching the bound (waiting for the release action to appear), the program may halt without releasing, leaving the resource unreleased. Note also that the suppression automaton cannot simply suppress resource acquisitions and uses because this would modify sequences that actually do satisfy the policy, contrary to the definition of precise enforcement. Hence, insertion automata can enforce some properties that suppression automata cannot.

For any suppression automaton, we can construct an insertion automaton that enforces the same property. The construction proceeds as follows. While the suppression automaton acts as a simple security automaton, the insertion automaton can clearly simulate it. When the suppression automaton decides to suppress an action a, it does so because there exists some extension σ of the input already processed (τ) such that $\hat{P}(\tau; \sigma)$ but $\neg\hat{P}(\tau; a; \sigma)$. Hence, when the suppression automaton suppresses a (giving up on precisely enforcing any sequence with $\sigma; a$ as a prefix), the insertion automaton merely inserts σ and terminates (also giving up on precise enforcement of sequences with $\sigma; a$ as a prefix). Of course, in practice, if σ is uncomputable or only intractably computable from τ, suppression automata are useful.

There are also many scenarios in which suppression automata are *more precise* enforcers than insertion automata. In particular, in situations such as the one described above in which we publish one policy but later need to restrict it due to changing system requirements or policy bugs, we can use suppression automata to suppress resource requests that are no longer allowed. Each suppression results in a new program action stream with an edit distance increased by 1, whereas the insertion automaton may produce an output with an arbitrary edit distance from the input.

Before we can characterize the properties that can be enforced by a suppression automaton, we must generalize our suppression functions so they act over sequences of symbols. Given a set of actions $\mathcal{A}$, a computable function $\omega^\star : \mathcal{A}^\star \to \mathcal{A}^\star$ is a *suppression function* if it satisfies the following conditions.

1. $\omega^\star(\cdot) = \cdot$
2. $\omega^\star(\sigma; a) = \omega^\star(\sigma); a$, or
   $\omega^\star(\sigma; a) = \omega^\star(\sigma)$

A suppression automaton can enforce the following properties.

[2] Premature termination of these executions takes us outside the system $\mathcal{S}$ since the rel symbol would be missing from the end of the sequence. To model the operation of a security automaton in such a situation we would need to separate the set of possible input sequences from the set of possible output sequences. For the sake of simplicity, we have not done so in this paper.

Theorem 6 (Nonuniform S-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ is enforceable by a suppression automaton if and only if there exists a suppression function $\omega^\star$ such that for all sequences $\sigma \in \mathcal{A}^\star$,*

- *if $\hat{P}(\sigma)$ then $\omega^\star(\sigma) = \sigma$, and*
- *if $\neg\hat{P}(\sigma)$ then*
  1. *$\forall\sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \neg\hat{P}(\sigma')$, or*
  2. *$\sigma \notin \Sigma$ and $\forall\sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \hat{P}(\omega^\star(\sigma'))$*

Limitations Similarly to its relatives, a suppression automaton is limited by the fact that some events may not be suppressible. For example, the program may have a direct connection to some output device and the automaton may be unable to interpose itself between the device and the program. It might also be the case that the program is unable to continue proper execution if an action is suppressed. For instance, the action in question might be an input operation.

3.3 Edit Automata

We form an *edit automaton* E by combining the insertion automaton with the suppression automaton. Our machine is now described by a 5-tuple with the form $(Q, q_0, \delta, \gamma, \omega)$. The operational semantics are derived from the composition of the operational rules from the two previous automata.

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \qquad \text{(E-STEPA)}$$

if $\sigma = a;\sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = +$

$$(\sigma, q) \xrightarrow{\cdot}_E (\sigma', q') \qquad \text{(E-STEPS)}$$

if $\sigma = a;\sigma'$
and $\delta(a, q) = q'$
and $\omega(a, q) = -$

$$(\sigma, q) \xrightarrow{\tau}_E (\sigma, q') \qquad \text{(E-INS)}$$

if $\sigma = a;\sigma'$
and $\gamma(a, q) = \tau, q'$

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \qquad \text{(E-STOP)}$$

otherwise

We again extend this single-step semantics to a multi-step semantics with the rules for reflexivity and transitivity.

Enforceable Properties As with insertion and suppression automata, edit automata are only capable of enforcing safety properties in uniform systems.

Theorem 7 (Uniform E-Enforcement) *If $\mathcal{S}$ is a uniform system and edit automaton E precisely enforces $\hat{P}$ on $\mathcal{S}$ then $\hat{P}$ obeys (*SAFETY*).*

The following theorem provides the formal basis for the intuition given above that insertion automata are strictly more powerful than suppression automata. Because insertion automata enforce a superset of properties enforceable by suppression automata, edit automata (which are a composition of insertion and suppression automata) precisely enforce exactly those properties that are precisely enforceable by insertion automata.

Theorem 8 (Nonuniform E-Enforcement) *A property $\hat{P}$ on the system $\mathcal{S} = (\mathcal{A}, \Sigma)$ can be enforced by some edit automaton if and only if there exists a function γ_p such that for all executions $\sigma \in \mathcal{A}^\star$, if $\neg\hat{P}(\sigma)$ then*

1. *$\forall\sigma' \in \Sigma.\sigma \prec \sigma' \Rightarrow \neg\hat{P}(\sigma')$, or*
2. *$\sigma \notin \Sigma$ and $\hat{P}(\sigma;\gamma_p(\sigma))$*

Although edit automata are no more powerful precise enforcers than insertion automata, we can very effectively enforce a wide variety of security policies conservatively with edit automata. We describe a particularly important application, the implementation of transactions policies, in the following section.

3.4 An Example: Transactions

To demonstrate the power of our edit automata, we show how to implement the monitoring of transactions. The desired properties of atomic transactions [EN94], commonly referred to as the ACID properties, are atomicity (either the entire transaction is executed or no part of it is executed), consistency preservation (upon completion of the transaction the system must be in a consistent state), isolation (the effects of a transaction should not be visible to other concurrently executing transactions until the first transaction is committed), and durability or permanence (the effects of a committed transaction cannot be undone by a future failed transaction).

The first property, atomicity, can be modeled using an edit automaton by suppressing input actions from the start of the transaction. If the transaction completes successfully, the entire sequence of actions is emitted atomically to the output stream; otherwise it is discarded. Consistency preservation can be enforced by simply verifying that the sequence to be emitted leaves the system in a consistent state. The durability or permanence of a committed transaction is ensured by the fact that committing a transaction is modeled by outputting the corresponding sequence of

actions to the output stream. Once an action has been written to the output stream it can no longer be touched by the automaton; furthermore, failed transactions output nothing. We only model the actions of a single agent in this example and therefore ignore issues of isolation.

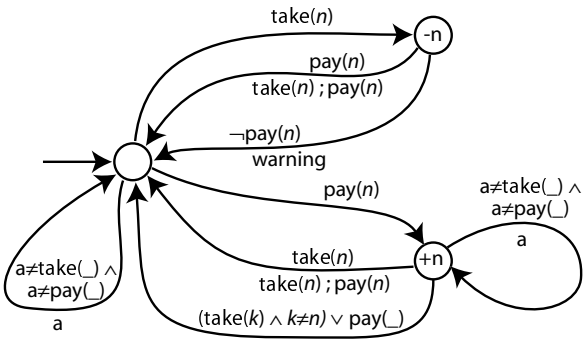


Figure 1: An edit automaton to enforce the market policy conservatively.

To make our example more concrete, we will model a simple market system with two main actions, take(n) and pay(n), which represent acquisition of n apples and the corresponding payment. We let a range over other actions that might occur in the system (such as window-shop or browse). Our policy is that every time an agent takes n apples it must pay for those apples. Payments may come before acquisition or vice versa. The automaton conservatively enforces atomicity of this transaction by emitting take(n); pay(n) only when the transaction completes. If payment is made first, the automaton allows clients to perform other actions such as browse before paying (the pay-take transaction appears atomically after all such intermediary actions). On the other hand, if apples are taken and not paid for immediately, we issue a warning and abort the transaction. Consistency is ensured by remembering the number of apples taken or the size of the prepayment in the state of the machine. Once acquisition and payment occur, the sale is final and there are no refunds (durability).

Figure 1 displays the edit automaton that conservatively enforces our market policy. The nodes in the picture represent the automaton states and the arcs represent the transitions. When a predicate above an arc is true, the transition will be taken. The sequence below an arc represents the actions that are emitted. Hence, an arc with no symbols below it is a suppression transition. An arc with multiple symbols below it is an insertion transition.

4 Future Work

We are considering a number of directions for future research.

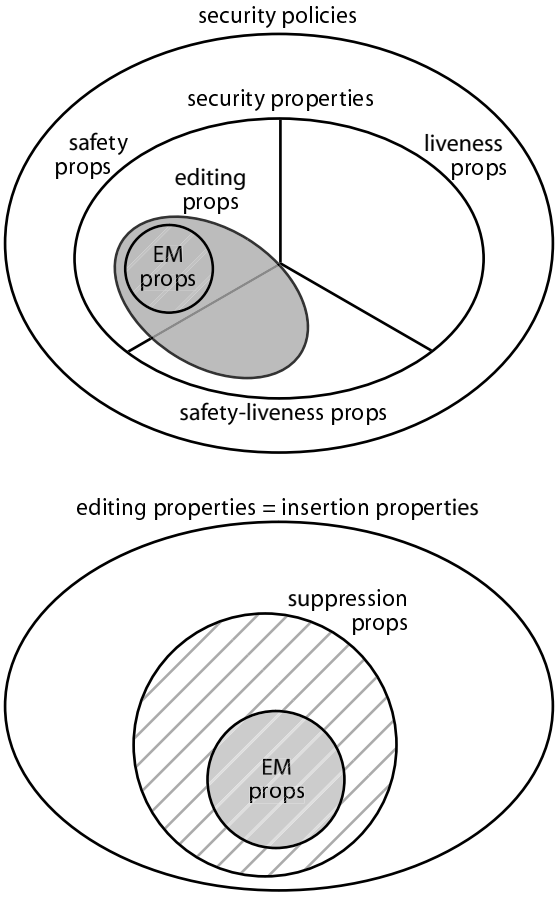


Figure 2: A taxonomy of *precisely* enforceable security policies.

Composing Schneider's security automata is straightforward [Sch00], but this is not the case for our edit automata. Since edit automata are sequence transformers, we can easily define the composition of two automata E_1 and E_2 to be the result of running E_1 on the input sequence and then running E_2 on the output of E_1. Such a definition, however, does not always give us the conjunction of the properties enforced by E_1 and E_2. For example, E_2 might insert a sequence of actions that violates E_1. When two automata operate on disjoint sets of actions, we can run one automaton after another without fear that they will interfere with one other. However, this is not generally the case. We are considering static analysis of automaton definitions to determine when they can be safely composed.

Our definitions of precise and conservative enforcement provide interesting bounds on the strictness with which properties can be enforced. Although precise enforcement of a property is most desirable because benign executions are guaranteed not to be disrupted by edits, disallowing even provably benign modifications restricts many useful transformations (for example, the enforcement of the mar-

ket policy from Section 3.4). Conservative enforcement, on the other hand, allows the most freedom in how a property is enforced because *every* property can be conservatively enforced by an automaton that simply halts on all inputs (by our assumption that $\hat{P}(\cdot)$). We are working on defining what it means to *effectively* enforce a property. This definition may place requirements on exactly what portions of input sequences must be examined, but will be less restrictive than precise enforcement and less general than conservative enforcement. Under such a definition, we hope to provide formal proof for our intuition that suppression automata effectively enforce some properties not effectively enforceable with insertion automata and vice versa, and that edit automata effectively enforce more properties than either insertion or suppression automata alone.

We found that edit automata could enforce a variety of non-EM properties on nonuniform systems, but could enforce no more properties on uniform systems than ordinary security automata. This indicates that it is essential to combine static program analysis and instrumentation with our new enforcement mechanisms if we wish to enforce the rich variety of properties described in this paper. We are very interested in exploring the synergy between these two techniques.

In practice, benign applications must be able to react to the actions of edit automata. When a program event is suppressed or inserted, the automaton must have a mechanism for signaling the program so that it may recover from the anomaly and continue its job (whenever possible). It seems likely that the automaton could signal an application with a security exception which the application can then catch, but we need experience programming in this new model.

We are curious as to whether or not we can further classify the space of security policies that can be enforced at runtime by constraining the resources available either to the running program or the run-time monitor. For example, are there practical properties that can be enforced by polynomial-time monitors but not linear-time monitors? What if we limit of the program's access to random bits and therefore its ability to use strong cryptography?

We are eager to begin an implementation of a realistic policy language for Java based on the ideas described in this paper, but we have not done so yet.

5 Conclusions

In this paper we have defined two new classes of security policies that can be enforced by monitoring programs at runtime. These new classes were discovered by considering the effect of standard editing operations on a stream of program actions. Figure 2 summarizes the relationship between the taxonomy of security policies discovered by Alpern and Schneider [AS87, Sch00] and our new editing properties.

Acknowledgment

The authors are grateful to Fred Schneider for making helpful comments and suggestions on an earlier version of this work.

Bibliography

[AS87] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[BLW02] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 54–66, Boston, January 2000. ACM Press.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[Lam85] Leslie Lamport. Logical foundation. *Lecture Notes in Computer Science*, 190:119–130, 1985.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[SS98] Anders Sandholm and Michael Schwartzbach. Distributed safety controllers for web services. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 1998.

[Thi01] Peter Thiemann. Enforcing security properties by type specialization. In *European Symposium on Programming*, Genova, Italy, April 2001.

[UES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, September 1999.

[UES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[Wal00] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

A Component Security Infrastructure

Yu David Liu and Scott F. Smith

Department of Computer Science
The Johns Hopkins University
{yliu, scott}@cs.jhu.edu

Abstract

This paper defines a security infrastructure for access control at the component level of programming language design. Distributed components are an ideal place to define and enforce significant security policies, because components are large entities that often define the political boundaries of computation. Also, rather than building a security infrastructure from scratch, we build on a standard one, the SDSI/SPKI security architecture [EFL+99].

1 Introduction

Today, the most widely used security libraries are lower-level network protocols, such as SSL, or Kerberos. Applications then need to build their own security policies based on SSL or Kerberos. What this ends up doing in practice is making the policy narrow and inflexible, limiting the design space of the implementation and thus the final feature set implemented for users. A component-level policy using a security infrastructure, on the other hand, can both (1) abstract away from keys and encryption to concepts of principal, certificate, and authorization; and, (2) abstract away from packets and sockets to service invocations on distributed components. This is the appropriate level of abstraction for most application-level programming, and creation of such an architecture is our goal.

It is well known that components [Szy98] are useful units upon which to place security policies. But, while there has been quite a bit of recent research integrating security into programming languages, little of this has been at the component level. Industry has been making progress securing components. The CORBA Security Specification [OMG02] layers existing security concepts and protocols on top of CORBA. It is designed for interoperability between components using different existing protocols such as Kerberos, SSL, etc, and not using a higher-level security architecture. Microsoft's DCOM uses a very simple ACL-based security layer.

In this paper we define a new component-layer security model built on top of the SDSI/SPKI security infrastructure [RL96, EFL+99, CEE+01]. By building on top of an existing security infrastructure we achieve two important gains: open standards developed by cryptographers are more likely to be correct than a new architecture made by us; and, a component security model built on SDSI/SPKI will allow the component security policies to involve other non-component SDSI/SPKI principals. We have chosen SDSI/SPKI in particular because it is the simplest general infrastructure which has been proposed as an Internet standard. Other architectures which could also be used to secure components include the PolicyMaker/KeyNote trust management systems [BFK99], and the Query Certificate Manager (QCM) extension [GJ00]. The most widely used PKI today, X.509 [HFPS99], is too hierarchical to secure the peer-to-peer interactions that characterize components.

Although we have found no direct precedent for our approach, there is a significant body of related work. PLAN is a language for active networks which effectively uses the PolicyMaker/KeyNote security infrastructure [HK99]; PLAN is not component-based. Several projects have generalized the Java Applet model to support mobile untrusted code [GMPS97, BV01, HCC+98]. These projects focus on mobile code, however, and not on securing distributed component connections; and, they do not use existing security infrastructures.

Components we informally characterize a software component. Our characterization here differs somewhat from the standard one [Szy98] in that we focus on the behavior of distributed components at *run-time* since that is what is relevant to dynamic access control.

1. Components are named, addressable entities, running at a particular location (take this to be a machine and a process on that machine);
2. Components have services which can be invoked;
3. Components may be distributed, *i.e.* services can be invoked across the network;

The above properties hold of the most widespread component systems today, namely CORBA, DCOM, and JavaBeans.

1.1 Fundamental Principles of Component Security

We now define some principles we believe should be at the core of secure component architectures. We implement these principles in our model.

Principle 9 *Each Component should be a principal in the security infrastructure, with its own public/private key pair.*

This is the most fundamental principle. It allows access control decisions to be made directly between components, not via proxies such as users, etc (but, users and organizations should also be involved in access control policies).

Principle 10 *As with other principals, components are known to outsiders by their public key.*

Public keys also serve as universal component names, since all components should have unique public keys.

Principle 11 *Components each have their own secured namespace for addressing other components (and other principals).*

By localizing namespaces to the component level, a more peer-to-peer, robust architecture is obtained. Components may also serve public names to outsiders.

Principle 12 *Components may be private—if they are not registered in nameservers, then since their public keys are not guessable, they are hidden from outsiders.*

This principle introduces a capability-like layer in the architecture: a completely invisible component is a secured component. Capabilities are also being used in other programming-language-based security models [Mil, vDABW96].

1.2 The Cell Component Architecture

We are developing a security architecture for a particular distributed component language, *Cells* [RS02]. A brief definition of cells is as follows.

> *Cells* are deployable containers of objects and code. They expose typed linking interfaces (*connectors*) that may import (*plugin*) and export (*plugout*) classes and operations. Via these interfaces, cells may be dynamically linked and unlinked, locally or across the network. Standard client-server style interfaces (*services*) are also provided for local or remote invocations. Cells may be dynamically loaded, unloaded, copied, and moved.

So, cells have run-time services like DCOM and CORBA components, but they also have *connectors* that allow for persistent connections. Persistent connections can be made across the Internet. This is particularly good for security because a connection is secured at set-up, not upon every transaction. This is analogous to SSH: once a secure session is in place, it can stay up for hours or days and perform many transactions without additional authorization checks. See Figure 1 for an example of two `Chatter` cells which are connected across the Internet via a persistent `Chat` connection. They can communicate by the `send` and `receive` operations.

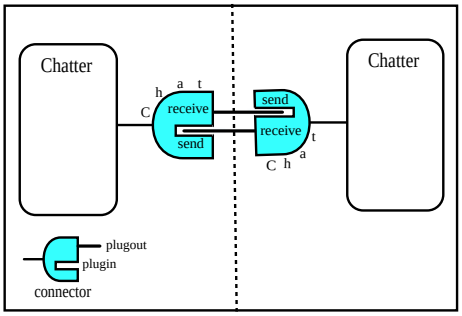


Figure 1: Two Chatter cells communicating over the network via Chat connector

There is a *president cell* in charge of cells at each location. The president can veto any access request and revoke any live connection at its location.

In this paper, we are not going to focus much on the cell-specific aspects, so the paper can be viewed as a proposal for a general component security architecture. The one exception is we do assume that each location has a single president cell which is in charge of all cells at that location, and even this assumption is not critical.

We use a simple, denotational model of cells here, which suffices for definition of the security model. We are currently implementing JCells, a modification of Java which incorporates the cell component architecture [Lu02].

1.3 The SDSI/SPKI Infrastructure

We very briefly review the SDSI/SPKI architecture, and give an overview of how we use it as a basis for the cell security architecture.

SDSI/SPKI is a peer-to-peer (P2P) architecture, a more flexible format than the centralized X.509 PKI [HFPS99]. Each SDSI/SPKI *principal* has a public/private key pair, and optional additional information such as its location and servers from which its certificate may be retrieved. *Certificates* are information signed by principals; forms of certificate include principals (giving the public key and optional information), group membership certificates (certifying a principal is a member of a certain group), and *authorization certificates* authorizing access. *Name servers* are

secure servers which map names (strings) to certificates. There is no *a priori* centralized structure to names—each server can have an arbitrary mapping. An *extended name* is a name which is a chain of name lookups; for instance asking one name server for "Joe's Sally" means to look up "Joe" on that name server, and then asking Joe's name server to look up "Sally". Access control decisions can be based either on direct lookup in an ACL, or via presentation of authorization certificates. In access control decisions using certificates, these certificates may be optionally delegated and revoked. See [RL96, EFL+99, CEE+01] for details.

We use SDSI/SPKI both for access control, and for a name service which allows cells to learn about each other securely. As mentioned above, each cell is registered as a SDSI/SPKI principal, and can issue certificates which fit the SDSI/SPKI format. Each cell can also serve SDSI/SPKI names. We currently don't generate certificates using the precise S-expression syntax of SDSI/SPKI, but the syntax is not an important issue (and, an XML format is probably preferred in any case). We incorporate SDSI/SPKI extended names, authorization certificates, delegation, and revocation models.

2 Cells

In this section we give a denotational model of cells and cell references (local or remote references to cells). Each cell has a public/private key and is thus a SDSI/SPKI principal. The public key not only serves for security, it serves to identify cells as no two cells should share a public key.

2.1 The Cell Virtual Machine (CVM)

The Cell Virtual Machine (CVM) is where cells run, in analogy to Java's JVM. The CVM is responsible for tasks including cell loading, unloading, serializing, deserializing and execution. Each CVM is represented by a particular cell, called its *president*. The president is a cell with extra service interfaces and connectors to implement the CVM's responsibilities. By reifying the CVM as a president cell, our architecture can be homogeneously composed of cells. It implicitly means CVM's are principals (identified by their president cells), which allows for CVM-wide security policies. In composing security policies, the president cell serves as a more abstract notion of location than the network-protocol-dependent locations used in existing security architectures such as Java's. This separation of low-level network protocols from the security architecture is particularly well-suited to mobile devices: even if network locations change from time to time, security policies can stay the same.

For the purposes of defining the security architecture, a simple denotational model of cells suffices: we ignore most of the run-time internals such as objects, classes and serialization, and focus on the access control policies for cells. We will define cells $c \in \mathbb{C}$ and cell references $cr \in \mathbb{CR}$. First we define the public and private keys of a cell.

2.2 Cell Identifiers CID and Locations LOC

All cells are uniquely identified by their *cell identifier, CID*.

Definition 1 *A cell identifier* $CID \in \mathbb{CID}$ *is the public key associated with a cell. A corresponding secret key* CID^{-1} *(*$\in \mathbb{CID}^{-1}$*) is also held by each cell.*

The *CID* is globally unique since it must serve as the identity of cells. All messages sent by a cell are signed by its CID^{-1}, and thus which will be verifiable given the *CID*.

When a cell is first loaded, its *CID* and CID^{-1} are automatically generated. Over the lifecycle of a cell, these values will not change regardless of any future unloading, loading, running, or migrating the cell may undergo. Since the *CID* is thus long-lived, it is sensible to make access control decisions based directly on cell identity.

Each loaded cell is running within a particular CVM. CVM's are located at network locations *LOC* which can be taken to be IP addresses.

2.3 Cell Denotations

We now define the denotation of a cell—each cell c is a structured collection of data including keys, certificates, etc. We don't define explicitly how cells evolve over time since that is not needed for the core security model definition.

Definition 2 *A cell* $\mathrm{c} \in \mathbb{C}$ *is defined as a tuple*

$$\mathrm{c} = \langle K, CertSTORE, SPT, NLT, BODY \rangle$$

where

- $K = \langle CID, CID^{-1} \rangle$ *are the cell's public and private keys.*
- $CertSTORE \in \mathbb{CERTSET}$ *is the set of certificates held by the cell for purposes of access and delegation.* $\mathbb{CERTSET}$ *is defined in Section 5.2 below.*
- $SPT \in \mathbb{SPT}$ *is the security policy table, defined in Section 5 below.*
- $NLT \in \mathbb{NLT}$ *is the naming lookup table, defined in Section 4 below.*
- $BODY$ *is the code body and internal state of the cell, including class definitions, service interfaces, connectors, and its objects. We treat this as abstract in this presentation.*

A cell environment $cenvt \in \mathbb{CENVT}$ is a snapshot of the state of all active cells in the universe: $\mathbb{CENVT} = \{C \subseteq \mathbb{C} \mid C$ finite and for any $c_1, c_2 \in C$, their keys differ$\}$.

2.4 Cell References

Cells hold *cell references*, cr, to other cells they wish to interact with. Structurally, a cell reference corresponds to a SDSI/SPKI principal certificate, including the cell's *CID* and possible location information. This alignment of SDSI/SPKI principal certificates with cell references is an important and fundamental aspect of our use of the SDSI/SPKI architecture. In an implementation, cell references will likely also contain cached information for fast direct access to cells, but we ignore that aspect here.

Definition 3 *A cell reference $cr \in \mathbb{CR}$ is defined as a tuple*

$$cr = \langle CID_{cell}, CID_{host}, LOC_{host} \rangle$$

where CID_{cell} is the CID of the referenced cell; CID_{host} is the CID of the CVM president cell where the referenced cell is located; LOC_{host} is the physical location of the CVM where the referenced cell is located. If cr refers to a CVM president cell, $CID_{cell} = CID_{host}$.

All interactions with cells by outsiders are through cell references; the information in cr can serve as a universal resource locator for cells since it includes the physical location and CVM in which the cell is running.

Definition 4 *$REF2CELL$ is defined as follows: If $cenvt \in \mathbb{CENVT}$ and $cr \in \mathbb{CR}$, $REF2CELL(cenvt, cr)$ returns the cell c which cr refers to.*

3 An Example

In this section we informally introduce our security architecture with an example modeled on a B2B (Business-to-Business) system such as a bookstore that trades online with multiple business partners.

In our example, there is a "Chief Trader" business that communicates with its trading partners Business A, Business B, and Business M, all of which are encapsulated as cells. Also, Business A has a partner Business K which is not a partner of the Chief Trader. Fig. 2 shows the layout of the cells and the CVM president cells that hold them. *CID*'s are large numbers and for conciseness here we abbreviate them as e.g. 4...9, hiding the middle digits. The source code for the Chief Trader has the following JCells code, which is more or less straightforward.

```
cell ChiefTrader {
  service IQuery {
    double getQuote(int No, String cate);
    List search(String condition);
  }
  connector ITrade {
    plugouts{
      boolean makeTrans(TradeInfo ti);
    }
    plugins{
      EndorseClass getEndorse();
    }
  }
 /* ... cell body here ... */
}
```

Name	CID	CVM	LOC
PtnrB	3...1	8...1	bbb.biz
CVM1	7...5	7...5	aaa.biz

Table 1a

Name	Extended Name
PtnrA	[CVM1,A]

Table 1b

Name	Group Members
PtnrGrp	{CR(CID=3...1, CVM=8...1, LOC=bbb.biz), PtnrA, [CVM1,M]}

Table 1c

Table 1: Naming Lookup Table for Chief Trader Cell

Name	CID	CVM	LOC
PtnrK	4...1	9...7	kkk.biz
ChiefTrader	1...3	6...5	chieftrader.biz

Table 2: Naming Lookup Table for Business A Cell

The major functionalities of Chief Trader are shown above. In service interface `IQuery`, `getQuote` takes the merchandise number and the category (promotion/adult's/children's book) and returns the quote; `search` accepts an SQL statement and returns all merchandise satisfying the SQL query. The Chief Trader cell also has an `ITrade` connector, which fulfills transactions with its business partners. Inside it, `getEndorse` is a plugin operation that needs to be implemented by the invoking cell, which endorses transactions to ensure non-repudiation.

Our infrastructure has the following features.

Non-Universal Names Cells can be universally identified by their *CID*'s. However, other forms of name are necessary to facilitate name sharing, including sharing by linked name spaces, described below. Instead of assuming the existence of global name servers, each cell contains a lightweight *Naming Lookup Table* (*NLT*), interpreting local names it cares about, and only those it cares about; example *NLT*'s are given in Tables 1, 2, and 3. An *NLT* entry maps a local name into one of three forms of value:a cell reference, extended name, or group. So, an *NLT* is three sub-tables for each sort; the three sub-tables for the Chief Trader are given in Figures 1a, 1b, and 1c, respectively. In Figure 1a, a local name is mapped to a cell reference $cr = \langle CID_{cell}, CID_{host}, LOC_{host} \rangle$: the cell with *CID* `3...1` running on a CVM on `bbb.biz` with president cell CID `8...1` is named `PtnrB` in the Chief Trader's namespace. In Figure 1b, a map from local names to *extended names* is defined, serving as an abbreviation. In Figure 1c a group `PtnrGrp` is defined. We now describe the meaning of these extended name and group name entries.

Extended Names Although names are local, namespaces of multiple cells can be linked together as a pow-

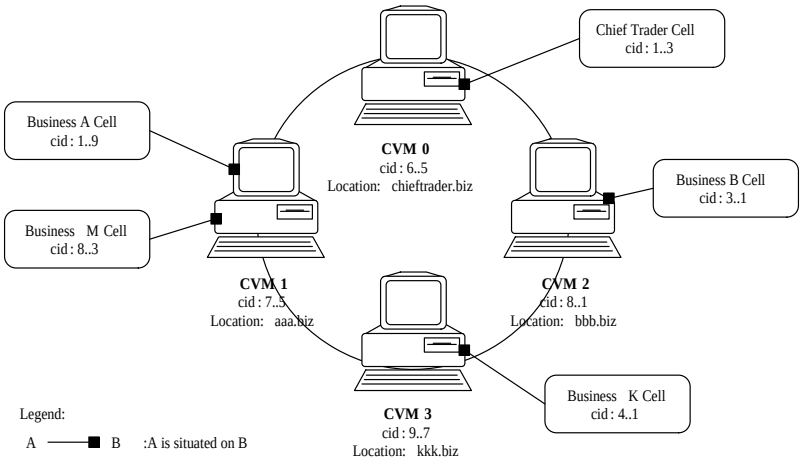


Figure 2: Example: A B2B System Across the Internet

Name	CID	CVM	LOC
A	1...9	thiscid	localhost
M	8...3	thiscid	localhost
Chief	1...3	6...5	chieftrader.biz

Table 3: Naming Lookup Table for CVM 1 President Cell

erful scheme to refer to other cells across cell boundaries; we just ran into an example, [CVM1, A] in Table 1c. This refers to a cell named A in the namespace of the cell named CVM1, from the perspective of the namespace of the Chief Trader cell itself. Extended names make it possible for a cell to interact with other cells whose *CID* and location information is not directly known: the Chief Trader cell does not directly keep track of the *CID* and location of Business A, but still has a name for it as PtnrA and the two can communicate. Extended names also help keep the size of naming information on any individual cell small.

Cell Groups Our system supports the definition of cell groups. The ability to define groups greatly eases the feasibility of defining security policies: policies can be group-based and need not be updated for every single cell. Table 1c defines a group PtnrGrp with three members. Group members can be direct cell references (CR(CID=3...1, CVM=8...1, LOC=bbb.biz)), local names (PtnrA), extended names ([CVM1,M]), or other sub-groups. This flexibility allows appropriate groups to be easily defined.

Access Control The security policy defines how resources should be protected. The units of cell protection include services, connectors and operations. Each cell contains a *security policy table* which specifies the precise access control policy. Table 4 gives the Chief Trader's security policy table. Here, operation search of its IQuery service can be invoked by any member in group PtnrGrp; the second entry in the table indicates that PtnrA can invoke any operation of its IQuery service; the third entry indicates PtnrA can also connect to its ITrade connector.

Hooks Security decisions can also be made contingent on the parameters passed in to an operation. This mechanism is called a *hook*, and provides access control at a finer level of granularity. The fourth entry of Table 4 has a hook h1 attached, which declares that the getQuote operation of IQuery service in the Chief Trader cell can be invoked by PtnrB only if the second parameter provided is equal to promotion, meaning only promotion merchandise can be quoted by PtnrB.

Delegation Delegation solves the problem of how two cells unaware of each other can interact effectively. For instance, Business K is not a partner of the Chief Trader, but since it is a partner of Business A (a partner of the Chief Trader), it is reasonable for Business K to conduct business with the Chief Trader. According to Table 4, Business A (named PtnrA in the name space of the Chief Trader cell) can have access to the IQuery interface of the Chief Trader, but business K can not. However, the Chief Trader has left the delegation bit (Delbit) of that entry to 1, which means any partner with this access right can delegate it to others. Table 5 shows Business A's security policy table; it defines a delegation policy whereby it grants PtnrK a certificate for the IQuery service of the Chief Trader cell. Thus cell K can invoke the IQuery service using this delegation certificate, even though the Chief Trader cell does not directly know Business K.

Subject	Resource	Accessright	Hook	Delbit
PtnrGrp	⟨thiscell, IQuery.search⟩	invoke	NULL	0
PtnrA	⟨thiscell, IQuery⟩	invoke	NULL	1
PtnrA	⟨thiscell, ITrade⟩	connect	NULL	0
PtnrB	⟨thiscell, IQuery.getQuote⟩	invoke	h1**	0

** h1(arg1, arg2) = {arg2 = "promotion"}

Table 4: Security Policy Table for Chief Trader

Subject	Resource	Accessright	Hook	Delbit
PtnrK	⟨ChiefTrader, IQuery⟩	invoke	NULL	0

Table 5: Security Policy Table for Business A

4 Name Services

Name services are needed to find cells for which a cell reference cr is lacking. The example of Section 3 gave several examples of name service in action.

4.1 Names and Groups

The Cell naming scheme uses the decentralized extended naming scheme of SPKI/SDSI [CEE+01]. In contrast with global naming schemes such as DNS and X.509 where global name servers are assumed, decentralized naming more reflects the nature of the Internet, for which Cells are designed. Each and every cell can serve as an independent, light-weight SDSI/SPKI naming server, so name service is pervasively distributed.

The set of local names are strings $n \in \mathbb{N}$. Local names map to cell references, cell extended names, and cell groups.

The extended name mechanism, illustrated in the example, enables a cell to refer to other cells through a chain of cross-cell name lookups.

Definition 5 *An extended name* $n_{ext} (\in \mathbb{N}_{EXT})$ *is a sequence of local names* $[n_1, n_2, \ldots, n_s]$.

Each n_{i+1} is a local name defined in the name space of the cell n_i.

SDSI/SPKI groups are represented as a name binding which maps the name of the group to the set of group members. Group members are themselves local names, which can in turn be mapped to cells or sub-groups. The owner of the group is the cell holding the local name.

Definition 6 *The set of groups* $\mathbb{G}$ *is defined as* $\mathbb{G} = POWER(\mathbb{CR} \cup \mathbb{N}_{EXT})$.

4.2 The Naming Lookup Table and Naming Interface

Local name bindings are stored in a *naming lookup table*, *NLT*. Every cell (including presidents) holds such a table, defining how local names relevant to it are mapped to cells and groups.

Definition 7 *Given a space* $\mathbb{V} = \mathbb{CR} \cup \mathbb{G} \cup \mathbb{N}_{EXT}$ *of values,*

1. *each naming lookup entry* $NLE \in \mathbb{NLE}$ *is a tuple* $NLE = \langle n, v \rangle$ *for* $n \in \mathbb{N}$ *and* $v \in \mathbb{V}$*;*
2. *a naming lookup table* $NLT \in \mathbb{NLT}$ *is a set of naming lookup entries:* $NLT \subseteq \mathbb{NLE}$ *such that for* $\langle n_1, v_1 \rangle, \langle n_2, v_2 \rangle \in NLT$, $n_1 \neq n_2$.

An example illuminating this definition can be found in Table 1 of Section 3. $GET\_VALUE(NLT, n)$ is a partial function which looks up the value of n in table *NLT*, and is undefined for names n with no mapping.

In the JCells language, the naming lookup table is maintained by an interface `INaming` present on all cells, which contains the operations for both looking up and modifying name lookup table information. Data in the naming lookup table can only be located or modified via `INaming`. The most important operation in `INaming` is `lookup`, which we now specify. Group membership is another important operation which we leave out of this short version.

4.3 Name Lookup

The `lookup`(*cenvt*, n_{ext}, cr_{srt}) operation, defined in Fig. 3, looks up an extended name n_{ext}, starting from the naming lookup table of the current cell which has reference cr_{srt}, and returns the final cell reference that n_{ext} refers to. Since arbitrary cell name servers could be involved in extended name lookup, the operation is parameterized by the global cell environment *cenvt*. A simple local name is a special case of an extended name with $LENGTH(n_{ext}) = 1$.

The first case in `lookup1` is when the name is not an extended name: the value v in the NLT is directly returned. The second case is for an extended name, and the next element of the extended name must be looked up using cell v's nameserver. The last case is where the value is an extended name itself.

The above algorithm does not define how the computation is distributed; to be clear, given an invocation `lookup` on a cell, the only parts involving distributed interaction are $REF2CELL$ and GET_VALUE, which in

```
lookup1(cenvt, n_ext, cr, pathset) =
        let [n_1, n_2, ..., n_s] = n_ext,
            ⟨K, _, _, NLT, _⟩ = REF2CELL(cenvt, cr),
            ⟨CID, CID_-1⟩ = K in
        if ⟨CID, n_1⟩ ∈ pathset then raise error;
        let pathset = pathset ∪ {⟨CID, n_1⟩} in
        v = GET_VALUE(NLT, n_1) in
                case v ∈ ℂℝ and LENGTH(n_ext) = 1: v
                case v ∈ ℂℝ and LENGTH(n_ext) > 1: lookup1(cenvt, [n_2, ..., n_s], v, pathset)
                case v ∈ 𝕄_EXT: let cr' = lookup1(cenvt, v, cr, pathset) in
                                lookup1(cenvt, [n_2, ..., n_s], cr', pathset)
                otherwise: raise error
lookup(cenvt, n_ext, cr) = lookup1(cenvt, n_ext, cr, φ)
```

Figure 3: Definition of *lookup*

combination locate another cell, look up a name from it and immediately return the value, be it a cell reference or extended name, to the initial invoker; all recursive calls to `lookup1` are local.

Compared with SDSI/SPKI [CEE+01], our algorithm is more general since it allows arbitary expansion of local names (the third case in the algorithm). For example, "Sallie's Joe's Pete" could in the process of resolution return an extended name for "Joe" on Sally as "Sam's Joe", which then must be resolved before "Pete" can be looked up on it; [CEE+01] will not resolve such names. However, this more general form is expressive enough that cyclic names are possible (e.g. "Sallie's Joe" maps to the extended name "Sallie's Joe's Fred") and a simple cycle detection algorithm must be used to avoid infinite computation. The cycle detection algorithm used in Fig. 3 is as follows. The `lookup` function maintains a set *pathset* of each recursive `lookup` name this initial request induces; if the same naming lookup entry is requested twice, a cycle is flagged and the algorithm aborted. A cycle can only be induced in the third case (local name expansion), where the algorithm divides the lookup process into two subtrees. Since there are only finitely many expansion entries possible, the algorithm always terminates.

5 Access Control

Access control decisions in the cell security architecture are based on the SDSI/SPKI model, specialized to the particular resources provided by cells. Each cell has associated with it a *security policy table*, which declares what *subjects* have access to what *resources* of the cell. This is an "object-based" use of SDSI/SPKI—every cell is responsible for controlling access to its resources.

Definition 8 *1. The set of subjects* $\mathbb{S} = \mathbb{N}_{EXT} \cup \{ALL\}$*: Subjects are extended names for cells or groups, or* `ALL` *which denotes any subject.*

2. The set of resources $\mathbb{R} = \langle \mathbb{O}, \mathbb{U} \rangle$*, where*

- $\mathbb{O} = \mathbb{N}_{EXT} \cup \{$`thiscell`$\}$ *is the set of resource owners, with* `thiscell` *denoting the cell holding the security policy itself.*
- $\mathbb{U} = \mathbb{SRV} \cup \mathbb{CNT} \cup \mathbb{OP}$ *is the set of protection units, which can be a cell service interface, connector, or operation, respectively.*

A partial order $\preccurlyeq_u$ is defined on protection units: $u_1 \preccurlyeq_u u_2$ if and only if u_1 is subsumed by u_2; details are omitted from this short version.

Access rights are $\mathbb{A} = \{$`invoke`, `connect`$\}$, where if $u \in \mathbb{CNT}$, *a* will be `connect`, and if $u \in \mathbb{SRV} \cup \mathbb{OP}$, *a* will be `invoke`.

Definition 9 *A security policy entry* $spe \in \mathbb{SPE}$ *is a tuple* $spe = \langle s, r, a, h, d \rangle$*, meaning access right* a *to resource* r *can be granted to subject* s*, if it passes the security hook* h*. This access right can be further delegated if* d *is set to 1. Specifically, we require*

- $s \in \mathbb{S}, r \in \mathbb{R}, a \in \mathbb{A}$.
- $h \in \mathbb{H}$ *is an optional security hook, a predicate which may perform arbitrary checking before access rights are granted (details are in Section 5.1 below). It is* ϵ *unless* $r = \langle o, u \rangle$*,* $o =$ `thiscell` *and* $u \in \mathbb{OP}$*. The set of security hooks that is associated with operation* $op \in \mathbb{OP}$ *is denoted* $\mathbb{H}_{op}$*.*
- $d \in \mathbb{D} = \{0, 1\}$ *is the delegation bit as per the SDSI/SPKI architecture. It defines how the access rights can be further delegated, detailed in Section 5.2 below.*

The Security policy table is then a set of security policies held by a cell: $SPT \in \mathbb{SPT} = POWER(\mathbb{SPE})$.

5.1 Security Hooks

The access control model above restricts access rights to cell connectors and service interfaces based on requesting principals. However, more expressiveness is needed in some situations. One obvious example is local name entries: How can one protect a single local name entry, *i.e.*, one particular invocation of `lookup`? Currently, we have a naming interface defined (see Section 4.2), but completely protecting operation `lookup` is too coarse-grained. For these cases we need a parameterized security policy, in which the policy is based on the particular arguments passed to an operation such as `lookup`. Security hooks are designed to fill this need. Hooks can be implemented in practice either as JCells code or written in some simple specification language; here we abstractly view them as predicates. The set of security hooks $\mathbb{H}_{op}$ contains verifying predicates that are being checked when the associated operation *op* is triggered.

Definition 10 *Given security policy entry* $spe = \langle s, \langle o, op\rangle, a, h_{op}, d\rangle$*, a hook* $h_{op} \in \mathbb{H}_{op}$ *is a predicate*

$$h_{op}(v_1, v_2, ...v_m)$$

where $v_1, v_2, ...v_m$ *are operation op parameters, and each* $v_i \in \mathbb{VAL}$*, for* $\mathbb{VAL}$ *an abstract set of values which includes cell references cr along with integers and strings.*

Access control security hooks are checked right before invocation of the associated operation, and the invocation can happen only if the security hook returns true.

5.2 Delegation

We use the SPKI/SDSI delegation model to support delegation in cell access control. A subject wishing access to a cell's resource can present a collection of certificates authorizing access to that cell. And, revocation certificates can nullify authorization certificates.

Definition 11 *An authorization certificate* $AuthC \in \mathbb{AUTHC}$ *is a signed tuple*

$$AuthC = \langle CID_I, CID_D, CID_R, u, a, d\rangle$$

where CID_I *is the CID of certificate issuer cell;* CID_D *is the CID of the cell being delegated;* CID_R *is the CID of resource owner cell;* u *is the resource unit;* a *is the access right to be delegated;* d *is the delegation bit: if 1, the cell specified by* CID_D *can further delegate the authorization to other cells.*

Definition 12 *A revocation certificate* $RevoC \in \mathbb{REVOC}$ *is a signed tuple*

$$RevoC = \langle CID_I, CID_D, CID_R, u, a\rangle$$

In this definition, CID_I is the *CID* of revocation certificate issuer cell; CID_D is the cell which earlier received an authorization certificate from the issuer cell but whose certificate is now going to be revoked; CID_R is the *CID* of resource owner cell; u is the resource unit; and, a is the access right to be revoked. The set of certificates is defined as $\mathbb{CERTSET} = POWER(\mathbb{AUTHC} \cup \mathbb{REVOC})$.

Support for delegation is reflected in the definition of a security policy entry, SPE: a delegation bit d is included. This bit is set to permit the authorized requester cell to further delegate the access rights to other cells. Cells can define delegation security policies by composing security policy entries for resources they do not own. This kind of entry takes the form $\langle s, \langle o, u\rangle, a, \texttt{NULL}, d\rangle$, with $o \neq$ `thiscell`. This denotes that an $AuthC$ granting access a to u of o can be issued if the requester is s. Notice security hooks are meaningless in this case.

The delegation proceeds as follows: suppose that on the resource cell side there is a security policy entry $spe = \langle s, \langle \texttt{thiscell}, u\rangle, a, h, 1\rangle$; cell s will be granted an $AuthC$ if it requests access a to unit u on the resource cell. Cells holding $AuthC$ can define their own security policies on how to further delegate the access rights to a third party, issuing another $AuthC$, together with the certificate passed from its own delegating source. This propagation can iterate. Cells automatically accumulate "their" pile of authorization certificates from such requests, in their own $CertSTORE$. When finally access to the resource is attempted, the requestor presents a chain of $AuthC$ which the resource cell will check to determine if the access right should be granted. Existence of a certificate chain is defined by predicate $EXISTS\_CERTCHAIN$, see Fig. 4. In the definition, operator *COUNTERPART* maps authorization certificates to their corresponding revocation certificate, and vice-versa. $\mathbf{B}_3$ in Figure 4 checks revocation certificates: if some cell revokes the *AuthC* it issued earlier, it sends a corresponding *RevoC* to the resource owner. When any cell makes a request to the resource and presents a series of *AuthC*, the resource holder will also check if any *RevoC* matches the *AuthC*.

5.3 `isPermitted`: The Access Control Decision

Each cell in JCells has a built-in security interface `ISecurity`, which contains a series of security-sensitive operations. The most important is `isPermitted` (see Fig. 5), which checks if access right $a_{req}(\in \mathbb{A})$ to resource unit $u_{req}(\in \mathbb{U})$ can be granted to subject $cr_{req}(\in \mathbb{CR})$. If $u_{req} \in \mathbb{OP}$, a list of arguments are provided by $arglist_{req}(\in \mathbb{ARGLIST})$ for possible checking by a hook. A set of authorization certificates $CertSet_{req}(\in \mathbb{CERTSET})$ may also be provided. The cell performing the check is $cr_{chk}(\in \mathbb{CR})$. The environment for all active cells is $cenvt(\in \mathbb{CENVT})$.

$EXISTS\_CERTCHAIN(cenvt, cr_{chk}, cr_{req}, u_{req}, a_{req}, CertSet_{req})=$

let $(CID_{chk}, t_1, t_2) = cr_{chk}$ **in**
let $(CID_{req}, t_1, t_2) = cr_{req}$ **in**
let $\langle t_1, CertSTORE_{chk}, t_2, t_3, t_4\rangle = REF2CELL(cenvt, cr_{chk})$ **in**
case $\mathbf{B}_1$ and $\mathbf{B}_2$ and $\mathbf{B}_3$: $\langle CID_{k_1}, u_{k_1}, a_{req}\rangle$
otherwise ϵ

where
$\mathbf{B}_1 = \exists Auth_1, ...Auth_n \in CertSet_{req}$ with

$$\begin{aligned} Auth_1 &= \langle CID_{chk}, CID_{k_1}, \langle CID_{chk}, u_{k_1}\rangle, a_{req}, 1\rangle \\ Auth_2 &= \langle CID_{k_1}, CID_{k_2}, \langle CID_{chk}, u_{k_2}\rangle, a_{req}, 1\rangle \\ &\vdots \\ Auth_n &= \langle CID_{k_{n-1}}, CID_{req}, \langle CID_{chk}, u_{k_n}\rangle, a_{req}, d\rangle \qquad (d \in \{0,1\}) \end{aligned}$$

$\mathbf{B}_2 = u_{k_n} \preccurlyeq_u u_{k_{n-1}} ... \preccurlyeq_u u_{k_1}$
$\mathbf{B}_3 = \forall auth \in \{Auth_1, ..., Auth_n\}, COUNTERPART(auth) \notin CertSTORE_{chk}$

Figure 4: Definition of *EXISTS_CERTCHAIN*

`isPermitted`$(cenvt, cr_{req}, u_{req}, a_{req}, arglist_{req}, CertSet_{req}, cr_{chk}) =$
let $(CID_{req}, t_1, t_2) = cr_{req}$ **in**
let $\langle t_1, t_2, SPT_{chk}, t_3, t_4\rangle = REF2CELL(cenvt, cr_{chk})$ **in**
$(\exists \langle s, \langle o, u\rangle, a, h, d\rangle \in SPT_{chk}.$ $\mathbf{B}_1$ **and** $\mathbf{B}_2$ **and** $\mathbf{B}_3$ **and** $\mathbf{B}_4)$
or
let $t = EXISTS\_CERTCHAIN(cenvt, cr_{chk}, cr_{req}, u_{req}, a_{req}, CertSet_{req})$ **in**
let $\langle CID_{req}, u_{req}, a_{req}\rangle = t$ for $t \neq \epsilon$ **in**
$(\exists \langle s, \langle o, u\rangle, a, h, d\rangle \in SPT_{chk}.$ $\mathbf{B}_1$ **and** $\mathbf{B}_2$ **and** $\mathbf{B}_3$ **and** $\mathbf{B}_4)$

where
$\mathbf{B}_1 = (a = a_{req})$
$\mathbf{B}_2 = (o =$ `thiscell`$)$ **and** $(u_{req} \preccurlyeq_u u)$
$\mathbf{B}_3 =$
case $ISGROUP(cenvt, s)$: `isMember`$(cenvt, cr_{req}, s, cr_{chk})$
case $ISCELL(cenvt, s)$:
let $\langle CID_s, CID_{host_s}, LOC_{host_s}\rangle =$ `lookup`$(cenvt, s, cr_{chk})$ **in** $(CID_s = CID_{req})$
$\mathbf{B}_4 = h(arglist_{req})$ **where** $u_{req} \in \mathbb{OP}$

Figure 5: Definition of *isPermitted*

`isPermitted` grants access either if there is a direct entry in the security policy table granting access, or if proper authorization certificates are presented. The first **or** case checks if there is directly an entry in the security policy table granting access. If authorization certificates are provided together with the request (second **or** case), permission will be granted if these certificates form a valid delegation chain, and the first certificate of the chain can be verified to be in the security policy table. $\mathbf{B}_1$ matches the access right; $\mathbf{B}_2$ matches the resource; $\mathbf{B}_3$ matches the subjects, which is complicated by the case a subject is a group; and $\mathbf{B}_4$ checks the security hook if any.

6 Conclusions

In this paper we have shown how the SDSI/SPKI infrastructure can be elegantly grafted onto a component architecture to give a general component security architecture. Particularly satisfying is how components can serve as principals, and how SDSI/SPKI naming gives a secure component naming system. We believe this infrastructure represents a good compromise between simplicity and expressivity. Very simple architectures which have no explicit access control or naming structures built-in lack the ability to express policies directly and so applications would need to create their own policies. More complex architectures such as trust management systems [BFK99] are difficult for everyday programmers to understand and thus may lead to more security holes.

Beyond our idea to use SDSI/SPKI for a peer-to-peer component security infrastructure, this paper makes several other contributions. We define four principles of component security, including the principle that components themselves should be principals. An implementation aspect developed here is the *cell reference*: the public key

plus location information is the necessary and sufficient data to interact with a cell. This notion combines programming language implementation needs with security needs: the cell needs to be accessed, and information supposedly from it needs to be authenticated. Modelling each CVM with a president cell simplifies the definition of per-site security policies. It also separates low-level location information from security polices, a structure well-suited to mobile devices. We define a name lookup algorithm which is more complete than the one given in [CEE+01]—extended names can themselves contain extended names, and all names can thus be treated uniformly in our architecture. Our architecture for name service is more pervasive than the distributed architecture proposed in SDSI/SPKI—every cell has its own local names and can automatically serve names to others. So while we don't claim any particularly deep results in this paper, we believe the proposal represents a simple, elegant approach that will work well in practice.

Many features are left out of this brief description. SDSI/SPKI principals that are not cells should be able to interoperate with cell principals. Several features of SDSI/SPKI and of cells are not modeled. We have not given many details on how data sent across the network is signed and encrypted.

The case of migrating cells is difficult and largely skipped in the paper; currently cells migrate with their private key, and a malicious host can co-opt such a cell. There should never simultaneously be two cells with the same *CID*, but since the system is open and distributed it could arise in practice. By making *CID*'s significantly long and being careful in random number generation, the odds of accidentally generating the same *CID* approach zero; more problematic is when a *CID* is explicitly reused, either by accident or through malicious intent. In this case a protocol is needed to recognize and resolve this conflict, a subject of future work.

Bibliography

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Security Protocols—6th International Workshop*, volume 1550 of *Lecture Notes in Computer Science*, pages 59–66. Springer-Verlag, 1999.

[BV01] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 4:359–384, 2001.

[CEE+01] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, pages 285–322, 2001.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. Internet Engineering Task Force RFC2693, September 1999. `ftp://ftp.isi.edu/in-notes/rfc2693.txt`.

[GJ00] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, December 1997.

[HCC+98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. RFC 2459: Internet X.509 public key infrastructure certificate and CRL profile, January 1999. `ftp://ftp.internic.net/rfc/rfc2459.txt`.

[HK99] Michael Hicks and Angelos D. Keromytis. A Secure PLAN. In *Proceedings of the First International Working Conference on Active Networks (IWAN '99)*, volume 1653, pages 307–314. Springer-Verlag, 1999.

[Lu02] Xiaoqi Lu. Report on the cell prototype project. (Internal Report), March 2002.

[Mil] Mark Miller. The E programming language. `http://www.erights.org`.

[OMG02] OMG. Corba security service specification, v1.8. Technical report, Object Management Group, March 2002. `http://www.omg.org/technology/documents/formal/security_service.htm`.

[RL96] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure, 1996. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[RS02] Ran Rinat and Scott Smith. Modular internet programming with cells. In *ECOOP 2002*, Lecture Notes in Computer Science. Springer Verlag, 2002. `http://www.cs.jhu.edu/hog/cells`.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[vDABW96] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.

Static Use-Based Object Confinement

Christian Skalka
The Johns Hopkins University
ces@cs.jhu.edu

Scott F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

Abstract

The confinement of object references is a significant security concern for modern programming languages. We define a language that serves as a uniform model for a variety of confined object reference systems. A *use-based* approach to confinement is adopted, which we argue is more expressive than previous communication-based approaches. We then develop a readable, expressive type system for static analysis of the language, along with a type safety result demonstrating that run-time checks can be eliminated. The language and type system thus serve as a reliable, declarative and efficient foundation for secure capability-based programming and object confinement.

1 Introduction

The confinement of object references is a significant security concern in languages such as Java. Aliasing and other features of OO languages can make this a difficult task; recent work [21, 4] has focused on the development of type systems for enforcing various containment policies in the presence of these features. In this extended abstract, we describe a new language and type system for the implementation of object confinement mechanisms that is more general than previous systems, and which is based on a different notion of security enforcement.

Object confimement is closely related to *capability*-based security, utilized in several operating systems such as EROS [16], and also in programming language (PL) architectures such as J-Kernel [6], E [5], and Secure Network Objects [20]. A capability can be defined as a reference to a data segment, along with a set of access rights to the segment [8]. An important property of capabilities is that they are *unforgeable*: it cannot be faked or reconstructed from partial information. In Java, object references are likewise unforgeable, a property enforced by the type sytem; thus, Java can also be considered a statically enforced capability system.

So-called *pure* capability systems rely on their high level design for safety, without any additional system-level mechanisms for enforcing security. Other systems *harden* the pure model by layering other mechanisms over pure capabilities, to provide stronger system-level enforcement of security; the `private` and `protected` modifiers in Java are an example of this. Types improve the hardening mechanisms of capability systems, by providing a declarative statement of security policies, as well as improving run-time efficiency through static, rather than dynamic, enforcement of security. Our language model and static type analysis focuses on capability hardening, with enough generality to be applicable to a variety of systems, and serves as a foundation for studying object protection in OO lanaguages.

2 Overview of the pop system

In this section, we informally describe some of the ideas and features of our language, called pop, and show how they improve upon previous systems. As will be demonstrated in Sect. 5, pop is sufficient to implement various OO language features, e.g. classes with methods and instance variables, but with stricter and more reliable security.

Use vs. communication-based security

Our approach to object confinement is related to previous work on containment mechanisms [2, 4, 21], but has a different basis. Specifically, these containment mechanisms rely on a *communication*-based approach to security; some form of barriers between objects, or domain boundaries, are specified, and security is concerned with communication of objects (or object references) across those boundaries. In our *use*-based approach, we also specify domain boundaries, but security is concerned with how objects are *used* within these boundaries. Practically speaking, this means that security checks on an object are performed when it is used (selected), rather than communicated.

The main advantage of the use-based approach is that security specifications may be more fine-grained; in a communication based approach we are restricted to a whole-object "what-goes-where" security model, while with a use-based approach we may be more precise in specifying what methods of an object may be used within various domains. Our use-based security model also allows "tunneling" of objects, supporting the multitude of protocols which rely on an intermediary that is not fully trusted.

$m, x, \varsigma, \text{set}, \text{get} \in ID,\ \iota \subseteq ID$			*identifiers*
$l \in Loc$			*locations*
$d \in \mathcal{D},\ D \subseteq \mathcal{D}$			*domains*
$\varphi \in \mathcal{D} \to 2^{ID}$			*interfaces*
ϱ	::=	$m_i(x) = e_i\ ^{0<i\leq n}$	*method lists*
co	::=	$[\varrho] \cdot d \mid l$	*core objects*
o	::=	$co \cdot \varphi \mid \mathbf{weak}_\iota(o)$	*object definitions*
v	::=	$x \mid o$	*values*
e	::=	$v \mid e.m(e) \mid e \upharpoonright (d, \iota) \mid \text{let } x = v \text{ in } e \mid \text{ref}_\varphi e$	*expressions*
E	::=	$[\,] \mid E.m(e) \mid v.m(E) \mid E \upharpoonright (d, \iota) \mid \text{ref}_\varphi E \mid \mathbf{weak}_\iota(E)$	*evaluation contexts*

Figure 1: Grammar for pop

Casting and weakening

Our language features a *casting* mechanism, that allows removal of access rights from particular views of an object, resulting in a greater attenuation of security when necessary. This casting discipline is statically enforced. It also features *weak* capabilities, a sort of deep-casting mechanism inspired by the same-named mechanism in EROS [16]. A weakened capability there is read-only, and any capabilities read from a weakened capability are automatically weakened. In our higher-level system, capabilities *are* objects, and any method access rights may be weakened.

Static protection domains

The pop language is an object-based calculus, where object methods are defined by lists of method definitions in the usual manner. For example, substituting the notation ... for syntactic details, the definition of a file object with read and write methods would appear as follows:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot \ldots \cdot \ldots$$

Additionally, every object definition statically asserts membership in a specific *protection domain* d, so that expanding on the above we could have:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot d \cdot \ldots$$

While the system requires that all objects are annotated with a domain, the *meaning* of these domains is flexible, and open to interpretation. Our system, considered in a pure form, is a core analysis that may be specialized for particular applications. For example, domains may be as interpreted as code owners, or they may be interpreted as denoting regions of static scope—e.g. package or object scope.

Along with domain labels, the language provides a method for specifying a security policy, dictating how domains may interact, via *user interface* definitions φ. Each object is annotated with a user interface, so that letting φ be an appropriately defined user interface and again expanding on the above, we could have:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot d \cdot \varphi$$

We describe user interfaces more precisely below, and illustrate and discuss relevant examples in Sect. 5.

Object interfaces

Other secure capability-based language systems have been developed [5, 6, 20] that include notions of access-rights interfaces, in the form of object types. Our system provides a more fine-grained mechanism: for any given object, its user-interface definition φ may be defined so that different domains are given more or less restrictive views of the same object, and these views are statically enforced. Note that the use-based, rather than communication-based, approach to security is an advantage here, since the latter allows us to more precisely modulate *how* an object may be used by different domains, via object method interfaces.

For example, imagining that the file objects defined above should be read-write within their local domain, but read only outside of it, an appropriate definition of φ for these objects would be as follows:

$$\varphi \triangleq \{d \mapsto \{\text{read}, \text{write}\}, \partial \mapsto \{\text{read}\}\}$$

The distinguished domain label ∂ matches any domain, allowing *default* interface mappings and a degree of "open-endedness" in program design.

The user interface is a mapping from domains to access rights—that is, to sets of methods in the associated object that each domain is authorized to use. This looks something like an ACL-based security model; however, ACLs are defined to map *principals* to privileges. Domains, on the other hand, are fixed boundaries in the code which may have nothing to do with principals, The practical usefulness of a mechanism with this sort of flexibility has been described in [3], in application to mobile programs.

$$d, ([\varrho] \cdot d' \cdot \varphi).m_i(v), \sigma \rightarrow d', (e_i[([\varrho] \cdot d' \cdot \varphi')/\varsigma])[v/x], \sigma \quad (\textit{send})$$
$$\text{where } \varrho = (m_j(x) = e_j\ ^{0<j\leq n}),\ 0 < i \leq n,\ m_i \in \varphi(d)$$
$$\text{and } \varphi' = \{d' \mapsto \{m_1, \ldots, m_n\}\}$$

$$d, (co \cdot \varphi) \triangleright (d', \iota), \sigma \rightarrow d, (co \cdot \varphi \oplus d' \mapsto \iota), \sigma \quad \iota \subseteq \varphi(d') \quad (\textit{cast})$$

$$d, \mathbf{weak}_\iota(o) \triangleright (d', \iota'), \sigma \rightarrow d, \mathbf{weak}_\iota(o \triangleright (d', \iota')), \sigma \quad (\textit{castweak})$$

$$d, \mathbf{weak}_\iota(o).m(v), \sigma \rightarrow d', \mathbf{weak}_\iota(e), \sigma' \quad (\textit{weaken})$$
$$\text{if } m \notin \iota \text{ and } d, o.m(v), \sigma \rightarrow d', e, \sigma'$$

$$d, \mathrm{ref}_\varphi v, \sigma \rightarrow d, l \cdot \varphi, \sigma \oplus l \mapsto v \quad l \notin \mathrm{dom}(\sigma) \quad (\textit{newcell})$$

$$d, (l \cdot \varphi).\mathrm{set}(v), \sigma \rightarrow d, v, \sigma \oplus l \mapsto v \quad \text{if set} \in \varphi(d) \quad (\textit{set})$$

$$d, (l \cdot \varphi).\mathrm{get}(), \sigma \rightarrow d, \sigma(l), \sigma \quad \text{if get} \in \varphi(d) \quad (\textit{get})$$

$$d, \mathrm{let}\ x = v\ \mathrm{in}\ e, \sigma \rightarrow d, e[v/x], \sigma \quad (\textit{let})$$

$$d, E[e], \sigma \rightarrow d', E[e'], \sigma' \quad \text{if } d, e, \sigma \rightarrow d', e', \sigma' \quad (\textit{context})$$

Figure 2: Operational semantics for pop

3 The language pop: syntax and semantics

We now formally define the syntax and operational semantics of pop, an object-based language with state and capability-based security features. The grammar for pop is defined in Fig. 1. It includes a countably infinite set of identifiers $\mathcal{D}$ which we refer to as *protection domains*. The definition also includes the notation $m_i(x) = e_i\ ^{0<i\leq n}$ as an abbreviation for $m_1(x) = e_1, \ldots, m_n(x) = e_n$; henceforth we will use this same vector abbreviation method for all language forms. Read-write cells are defined as primitives, with a cell constructor $\mathrm{ref}_\varphi v$ that generates a read-write cell containing v, with user interface φ. The object weakening mechanism $\mathbf{weak}_\iota(o)$ described in the previous section is also provided, as is a casting mechanism $o \triangleright (d, \iota)$, which updates the interface φ associated with o to map d to ι. The operational semantics will ensure that only *downcasts* are allowed.

We require that for any φ and d, the method names $\varphi(d)$ are a subset of the method names in the associated object. Note that object method definitions may contain the distinguished identifier ς which denotes *self*, and which is bound by the scope of the object; objects always have full access to themselves via ς. We require that self never appear "bare"—that is, the variable ς must always appear in the context of a method selection $\varsigma.m(e)$. This restriction ensures that ς cannot escape its own scope, unintentially providing a "back-door" to the object. *Rights amplification* via ς is still possible, but this is a *feature* of capability-based security, not a flaw of the model.

The small-step operational semantics for pop is defined in Fig. 2 as the relation $\rightarrow$ on *configurations* d, e, σ, where *stores* σ are partial mapping from locations l to values v. The reflexive, transitive closure of $\rightarrow$ is denoted $\rightarrow^\star$. If $d_1, e, \varnothing \rightarrow^\star d', e', \sigma'$ with d_1 the top-level domain for all all programs, then if e' is a value we say e *evaluates to* e', and if e' is not a value and d', e', σ' cannot be reduced, then e is said to *go wrong*. If $x \in \mathrm{dom}(f)$, the notation $f \oplus x \mapsto v$ denotes the function which maps x to v and otherwise is equivalent to f. If $x \notin \mathrm{dom}(f)$, $f \oplus x \mapsto v$ denotes the function which extends f, mapping x to v. All interfaces φ are *quasi*-constant, that is, constant except on a known finite set D (since they're statically user-defined with default values), and we write $\varphi \oplus \partial \mapsto \iota$ to denote φ' which has default value ι, written $\varphi'(\partial)$, and where $\varphi'(d) = \varphi(d)$ for all $d \in D$.

In the *send* rule, full access to self is ensured via the object that is substituted for ς; note that this is the selected object o, updated with an interface φ' that confers full access rights on the domain of o. The syntactic restriction that ς never appear bare ensures that this strengthened object never escapes its own scope.

Of particular note in the semantics is the myriad of runtime security checks associated with various language features; our static analysis will make these uneccessary, by compile-time enforcement of security.

4 Types for pop: the transformational approach

To obtain a sound type system for the pop language, we use the *transformational* approach; we define a semantics-preserving transformation of pop into a *target* language, that comes pre-equipped with a sound let-polymorphic type system. This technique has several advantages: since the transformation is computable and easy to prove correct, a sound *indirect* type system for pop can be obtained as the composition of the transformation and type judge-

$$
\begin{array}{rcl}
\{d_1 \mapsto \iota_1, \cdots, \widehat{d_n \mapsto \iota_n}, \partial \mapsto \iota\} & = & \{\varnothing\}\{\partial = \iota\}\{d_1 = \iota_1\}\ldots\{d_n = \iota_n\} \\
[\![x]\!]_d & = & x \\
[\![\varsigma]\!]_d & = & (\varsigma\{\}) \\
[\![[m_i(x) = e_i\ {}^{0<i\leq n}] \cdot d' \cdot \varphi]\!]_d & = & \text{let } o_\varsigma = \mathsf{fix}\,\varsigma.\lambda\_.\{\text{obj} = \{m_i = \lambda x.[\![e_i]\!]_{d'}\ {}^{0<i\leq n}\}, \\
 & & \qquad\qquad \text{ifc} = \{d' = \{m_1, \ldots, m_n\}\}, \\
 & & \qquad\qquad \text{strong} = \bar{\varnothing}\}\ \text{in} \\
 & & \{\text{obj} = (o_\varsigma\{\}).\text{obj}, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\
[\![e_1.m(e_2)]\!]_d & = & \text{let } c_1 = [\![e_1]\!]_d \text{ in} \\
 & & c_1.\text{strong} \ni m; \\
 & & (c_1.\text{ifc}.d \vee c_1.\text{ifc}.\partial) \ni m; \\
 & & \text{let } c_2 = (c_1.\text{obj}.m)([\![e_2]\!]_d) \text{ in} \\
 & & \text{let } w = c_1.\text{strong} \wedge c_2.\text{strong} \text{ in} \\
 & & \{\text{obj} = c_2.\text{obj}, \text{ifc} = c_2.\text{ifc}, \text{strong} = w\} \\
[\![e_! (d', \{m_1, \ldots, m_n\})]\!]_d & = & \text{let } c = [\![e]\!]_d \text{ in} \\
 & & c.\text{ifc}.d' \ni m_1; \ldots; c.\text{ifc}.d' \ni m_n; \\
 & & \text{let } i = (c.\text{ifc})\{d' = \{m_1, \ldots, m_n\}\} \text{ in} \\
 & & \{\text{obj} = c.\text{obj}, \text{ifc} = i, \text{strong} = c.\text{strong}\} \\
[\![\mathbf{weak}_\iota(e)]\!]_d & = & \text{let } c = [\![e]\!]_d \text{ in} \\
 & & \{\text{obj} = c.\text{obj}, \text{ifc} = c.\text{ifc}, \text{strong} = c.\text{strong} \ominus \iota\} \\
[\![\text{ref}_\varphi e]\!]_d & = & \text{let } x = \text{ref } [\![e]\!]_d \text{ in} \\
 & & \text{let } o = \{\text{get} = \lambda y.!x, \text{set} = \lambda y.x := y\} \text{ in} \\
 & & \{\text{obj} = o, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\
[\![\text{let } x = e_1 \text{ in } e_2]\!]_d & = & \text{let } x = [\![e_1]\!]_d \text{ in } [\![e_2]\!]_d
\end{array}
$$

Figure 3: The pop-to-pml term transformation

ments in the target language, eliminating the overhead of a type soundess proof entirely. The technique also eases development of a *direct* pop type system— that is, where pop expressions are treated directly, rather than through transformation. This is because safety in a direct system can be demonstrated via a simple proof of correspondance between the direct and indirect type systems, rather than through the usual (and complicated) route of subject reduction. This technique has been used to good effect in previous static treatments of languages supporting stack-inspection security [12], information flow security [11], and elsewhere [15].

While we focus on the logical type system in this presentation, we will briefly describe how the transformational approach has benefits for type *inference*, allowing an algorithm to be developed using existing, efficient methods.

4.1 The pop-to-pml transformation

The target language of the transformation is pml [18, 19], a calculus of extensible records based on Rémy's Projective ML [13], and equipped with references, sets and set operations. The language pml allows definition of records with default values $\{v\}$, where for every label a we have $\{v\}.a = v$. Records r may be modified with the syntax $r\{a = v\}$, such that $(r\{a = v\}).a = v$ and $(r\{a = v\}).a' = r.a'$ for all other a'.

The language also allows definition of finite sets B of atomic identifiers b chosen from a countably infinite set $\mathcal{L}_b$, and cosets $\bar{B}$. This latter feature presents some practical implementation issues, but in this presentation we take it at mathematical face value— that is, as the countably infinite set $\mathcal{L}_b - B$. The language also contains set operations $\ni$, $\vee$, $\wedge$ and $\ominus$, which are membership check, union, intersection and difference operations, respectively.

The pop-to-pml term transformation is given in Fig. 3. For brevity in the transformation we define the following syntactic sugar:

$$
\begin{array}{c}
\{m_1 = e_1, \ldots, m_n = e_n\} \\
\triangleq \\
\{\varnothing\}\{m_1 = e_1\}\cdots\{m_n = e_n\}
\end{array}
$$

$$\begin{array}{rcll} \tau & ::= & \alpha, \beta, \ldots \mid \tau \to \tau \mid \{\tau\} \mid a : \tau\,;\ \tau \mid \partial\tau \mid b\tau, \tau \mid \varnothing \mid \omega \mid \tau \text{ ref} \mid c & \textit{types} \\ c & ::= & + \mid - & \textit{constructors} \end{array}$$

Figure 4: pml type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau : \mathit{Type}}{\tau \text{ ref} : \mathit{Type}} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}_\varnothing, \mathit{Set}_\varnothing}{\{\tau\} : \mathit{Type}} \qquad \varnothing : \mathit{Set}_B \qquad \omega : \mathit{Set}_B \qquad c : \mathit{Con}$$

$$\frac{\tau : \mathit{Con} \qquad b \notin B \qquad \tau' : \mathit{Set}_{B \cup \{b\}}}{(b\tau, \tau') : \mathit{Set}_B} \qquad \frac{\tau : \mathit{Type}}{\partial\tau : \mathit{Row}_A} \qquad \frac{\tau : \mathit{Type} \qquad a \notin A \qquad \tau' : \mathit{Row}_{A \cup \{a\}}}{(a : \tau\,;\ \tau') : \mathit{Row}_A}$$

Figure 5: Kinding rules for pml types

$$\begin{array}{rcll} \mathsf{fix}\,\varsigma.\lambda\_.e & \triangleq & \mathsf{fix}\,\varsigma.\lambda x.e & x \text{ not free in } e \\ e_1; e_2 & \triangleq & \text{let } x = e_1 \text{ in } e_2 & x \text{ not free in } e_2 \end{array}$$

The translation is effected by transforming pop objects into rows with obj fields containing method transformations, ifc fields containing interface transformations, and strong fields containing sets denoting methods on which the object is *not* weak. Interface definitions φ are encoded as records $\hat{\varphi}$ with fields indexed by domain names; elevated rows are used to ensure that interface checks are total in the image of the transformation.

Of technical interest is the use of lambda abstractions with recursive binding mechanisms in pml— of the form $\mathsf{fix}\,z.\lambda x.e$, where z binds to $\mathsf{fix}\,z.\lambda x.e$ in e— to encode the self variable ς in the transformation. Also of technical note is the manner in which weakenings are encoded. In a pop weakened object $\mathbf{weak}_\iota(o)$, the set ι denotes the methods which are inaccessible via weakening. In the encoding these sets are turned "inside out", with the strong field in objects denoting the fields which *are* accessible. We define the translation in this manner to allow a simple, uniform treatment of set subtyping in pop; the following section elaborates on this.

The correctness of the transformation is established by the simple proof of the following theorem:

Theorem 4.1 (Transformation correctness) *If e evaluates to v then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_{d_1}$. If e diverges then so does $[\![e]\!]_{d_1}$. If e goes wrong then $[\![e]\!]_{d_1}$ goes wrong.*

4.2 Types for pop

A sound polymorphic type system for pml is obtained in a straightforward manner as an instantiation of $\mathrm{HM}(X)$ [9, 17], a constraint-based polymorphic type framework. Type judgements in $\mathrm{HM}(X)$ are of the form $C, \Gamma \vdash e : \sigma$, where C is a type constraint set, Γ is a typing environment, and σ is a polymorphic type scheme. The instantiation consists of a type language including row types [13] and a specialized language of *set* types, defined in Fig. 4. To ensure that only meaningful types can be built, we immediately equip this type language with *kinding* rules, defined in Fig. 5, and hereafter consider only well-kinded types. Note in particular that these kinding rules disallow duplication of record field and set element labels.

Set types behave in a manner similar to row types, but have a succinct form more appropriate for application to sets. In fact, set types have a direct interpretation as a particular form of row types [19], which is omitted here for brevity. The field constructors $+$ and $-$ denote whether a set element is present or absent, respectively. The set types $\varnothing$ and ω behave similarly to the uniform row constructor $\partial\tau$; the type $\varnothing$ (resp. ω) specifies that all other elements not explicitly mentioned in the set type are absent (resp. present). For example, the set $\{b_1, b_2\}$ has type $\{b_1+, b_2+, \varnothing\}$, while $\{\overline{b_1, b_2}\}$ has type $\{b_1-, b_2-, \omega\}$. The use of element and set variables γ and β allows for fine-grained polymorphism over set types.

Syntactic type safety for pml is easily established in the $\mathrm{HM}(X)$ framework [17]. By virtue of this property and Theorem 4.1, a sound, indirect static analysis for pop is immediately obtained by composition of the pop–to–pml transformation and pml type judgments:

Theorem 4.2 (Indirect type safety) *If e is a closed* pop *expression and $C, \Gamma \vdash [\![e]\!]_{d_1} : \sigma$ is valid, then e does not go wrong.*

While this indirect type system is a sound static analysis for pop, it is desirable to define a direct static analysis for pop. The term transformation required for the indirect analysis is an unwanted complication for compilation, the indirect type system is not a clear declaration of program properties for the programmer, and type error reporting would be extremely troublesome. Thus, we define a direct type system for pop, the development of which significantly benefits from the transformational approach. In

$$\tau ::= \alpha, \beta, \ldots \mid \{\tau\} \mid [\tau]^{\tau}_{\tau} \mid m : \tau \to \tau\,;\ \tau \mid d : \tau\,;\ \tau \mid m, \tau \mid \epsilon$$

Figure 6: Direct pop type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \epsilon : \mathit{Meth}_M, \mathit{Set}_M, \mathit{Ifc}_D \qquad \frac{m \notin M \qquad \tau : \mathit{Set}_{M \cup \{m\}}}{m, \tau : \mathit{Set}_M} \qquad \frac{\tau_1 : \mathit{Set}_{\varnothing} \qquad d \notin D \qquad \tau : \mathit{Ifc}_{D \cup \{d\}}}{d : \{\tau_1\}\,;\ \tau_2 : \mathit{Ifc}_D}$$

$$\frac{\tau_1 : \mathit{Type} \qquad \tau_2 : \mathit{Type} \qquad m \notin M \qquad \tau : \mathit{Meth}_{M \cup \{m\}}}{m : \tau_1 \to \tau_2\,;\ \tau : \mathit{Meth}_M} \qquad \frac{\tau_1 : \mathit{Meth}_{\varnothing} \qquad \tau_2 : \mathit{Ifc}_{\varnothing} \qquad \tau_3 : \mathit{Set}_{\varnothing}}{[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} : \mathit{Type}}$$

Figure 7: Direct pop type kinding rules

particular, type safety for the direct system may be demonstrated by a simple appeal to safety in the indirect system, rather than *ab initio*.

The direct type language for pop is defined in Fig. 6. We again ensure the construction of only meaningful types via kinding rules, defined in Fig. 7, hereafter considering only well-kinded pop types. The most novel feature of the pop type language is the form of object types $[\tau]^{\{\tau_1\}}_{\{\tau_2\}}$, where τ_2 is the type of any weakening set imposed on the object, and τ_1 is the type of its interface. Types of sets are essentially the sets themselves, modulo polymorphic features; we abbreviate a type of the form $\tau; \epsilon$ or τ, ϵ as τ.

The close correlation between the direct and indirect type system begins with the type language: types for pop have a straightforward interpretation as pml types, defined in Fig. 8. This interpretation is extended to constraints and typing environments in the obvious manner. In this interpretation, we turn weakening sets "inside-out", in keeping with the manner in which weakening sets are turned inside-out in the pop-to-pml term transformation. The benefit of this approach is with regard to subtyping: weakening sets can be safely strengthened, and user interfaces safely weakened, in a uniform manner via subtyping coercions.

The direct type judgement system for pop, the rules for which are *derived* from pml type judgements for transformed terms, is defined in Fig. 9. Note that subtyping in the direct type system is defined in terms of the type interpretation, where $C_1 \Vdash C_2$ means that every solution of C_1 is also a solution of C_2. The following definition simplifies the statement of the SEND rule:

Definition 4.1 *$C \Vdash m \notin \tau_w$ holds iff $(\!|C|\!) \not\Vdash \exists\phi.((\!|\tau_w|\!)_+ \leq (\!|m, \phi|\!)_+)$ holds, where $\phi \notin \mathrm{fv}(C, \tau_w)$.*

The easily proven, tight correlation between the indirect and direct pop type systems is clearly demonstrated via the following lemma:

Lemma 4.1 *$d, C, \Gamma \vdash e : \tau$ is valid iff $(\!|C|\!), (\!|\Gamma|\!) \vdash [\![e]\!]_d : (\!|\tau|\!)$ is.*

And in fact, along with Theorem 4.1, this correlation is sufficient to establish direct type safety for pop:

Theorem 4.3 (Direct type safety) *If e is a closed* pop *expression and $d, C, \Gamma \vdash e : \tau$ is valid, then e does not go wrong.*

This result demonstrates the advantages of the transformational method, which has allowed us to define a direct, expressive static analysis for pop with a minimum of proof effort.

An important consequence of this result is the implication that certain *optimizations* may be effected in the pop operational semantics, as a result of the type analysis. In particular, since the result shows that any well-typed program will be operationally safe, or *secure*, the various runtime security checks— i.e. those associated with the *send*, *cast*, *weaken*, *set* and *get* rules— may be eliminated entirely.

Type inference

The transformational method allows a similarly simplified approach to the development of type inference. The $\mathrm{HM}(X)$ framework comes equipped with a type inference algorithm modulo a constraint normalization procedure (constraint normalization is the same as constraint satisfaction, e.g. *unification* is a normalization procedure for equality constraints). Furthermore, efficient constraint normalization procedures have been previously developed for row types [10, 14], and even though set types are novel, their interpretation as row types [19] allows a uniform implementation. This yields a type inference algorithm for pml in the $\mathrm{HM}(X)$ framework. An indirect inference analysis for pop may then be immediately obtained as the composition of the pop-to-pml transformation and pml type inference.

Furthermore, a direct type inference algorithm can be derived from the indirect algorithm, just as direct type judgements can be derived from indirect judgements. Only

$$
\begin{array}{rcl}
(\!|[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}}|\!) & = & \{\text{obj} : \{(\!|\tau_1|\!)\}\,;\ \text{ifc} : \{(\!|\tau_2|\!)\}\,;\ \text{strong} : \{(\!|\tau_3|\!)_-\}\,;\ \partial\{\varnothing\}\} \\
(\!|m : \tau_1 \to \tau_2\,;\ \tau|\!) & = & m : (\!|\tau_1|\!) \to (\!|\tau_2|\!)\,;\ (\!|\tau|\!) \\
(\!|d : \{\tau_1\}\,;\ \tau_2|\!) & = & d : \{(\!|\tau_1|\!)_+\}\,;\ (\!|\tau_2|\!) \\
(\!|\epsilon|\!) & = & \partial\{\varnothing\} \\
(\!|\beta|\!), (\!|\beta|\!)_+, (\!|\beta|\!)_- & = & \beta \\
(\!|m, \tau|\!)_+ & = & m+, (\!|\tau|\!)_+ \\
(\!|\epsilon|\!)_+ & = & \varnothing \\
(\!|m, \tau|\!)_- & = & m-, (\!|\tau|\!)_- \\
(\!|\epsilon|\!)_- & = & \omega
\end{array}
$$

Figure 8: The pop-to-pml type transformation

the syntactic cases need be adopted, since efficient constraint normalization procedures for row types may be reused in this context— recall that the direct pop type language has a simple interpretation in the pml type language.

5 Using pop

By choosing different naming schemes, a variety of security paradigms can be effectively and reliably expressed in pop. One such scheme enforces a strengthened meaning of the `private` and `protected` modifiers in class definitions, a focus of other communication-based capability type analyses [4, 21]. As demonstrated in [21], a `private` field can leak by being returned by reference from a `public` method. Here we show how this problem can be addressed in a use-based model. Assume the following Java-like pseudocode package p, containing class definitions c_1, c_2, and possibly others, where c_2 specifies a method m that leaks a `private` instance variable:

```
package p begin

  class c1 {                    class c2 {
    public:                       public:
      f(x) = x                      m(x) = b
    private:                        a = new c1
      g(x) = x                  private:          ...
    protected:                      b = new c1
      h(x) = x                  protected:
  }                                 c = new c1
                                }
end
```

We can implement this definition as follows. Interpreting domains as class names in pop, let p denote the set of all class names $c_1, \ldots, c_n$ in package p, and let $p \mapsto \iota$ be syntactic sugar for $c_1 \mapsto \iota_1, \ldots, c_n \mapsto \iota_n$. Then, the appropriate interface for objects in the encoding of class c_1 is as follows:

$$\varphi_1 \triangleq \{p \mapsto \{f, h\}, \partial \mapsto \{f\}\}$$

(Recall that all objects automatically have full access to themselves, so full access for c_1 need not be explicitly stated). The class c_1 can then be encoded as an object *factory*, an object with only one publicly available method that returns new objects in the class, and some arbitrary label d:

$$
\begin{array}{rcl}
o_1 & \triangleq & [f(x) = x,\ g(x) = x,\ h(x) = x] \cdot c_1 \cdot \varphi_1 \\
\text{fctry}_{c_1} & \triangleq & [\text{new}(x) = o_1] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\}
\end{array}
$$

To encode c_2, we again begin with the obvious interface definition for objects in the encoding of class c_2:

$$\varphi_2 \triangleq \{p \mapsto \{m, a, c\}, \partial \mapsto \{m, a\}\}$$

However, we must now encode *instance variables*, in addition to methods. In general, this is accomplished by encoding instance variables a containing objects as methods $a()$ that return references to objects. Then, any selection of a is encoded as $a()$.get(), and any update with v is encoded $a()$.set(v). By properly constraining the interfaces on these references, a "Java-level" of modifier enforcement can be achieved; but casting the interfaces of stored objects *extends* the security, by making objects *unusable* outside the intended domain. Let $e \upharpoonright (\{d_1, \ldots, d_n\}, \iota)$ be sugar for $e \upharpoonright (d_1, \iota) \upharpoonright \cdots \upharpoonright (d_n, \iota)$. Using fctry_{c_1}, we may create a `public` version of an object equivalent to o_1, without any additional constraints on its confinement, as follows:

$$o_a \triangleq \text{fctry}_{c_1}.\text{new}()$$

Letting $p' = p - \{c_2\}$, we may create a version of an object equivalent to o that is `private` with respect to the encoding of class c_2, using casts as follows:

$$o_b \triangleq (\text{fctry}_{c_1}.\text{new}()) \upharpoonright (\partial, \varnothing) \upharpoonright (p', \varnothing)$$

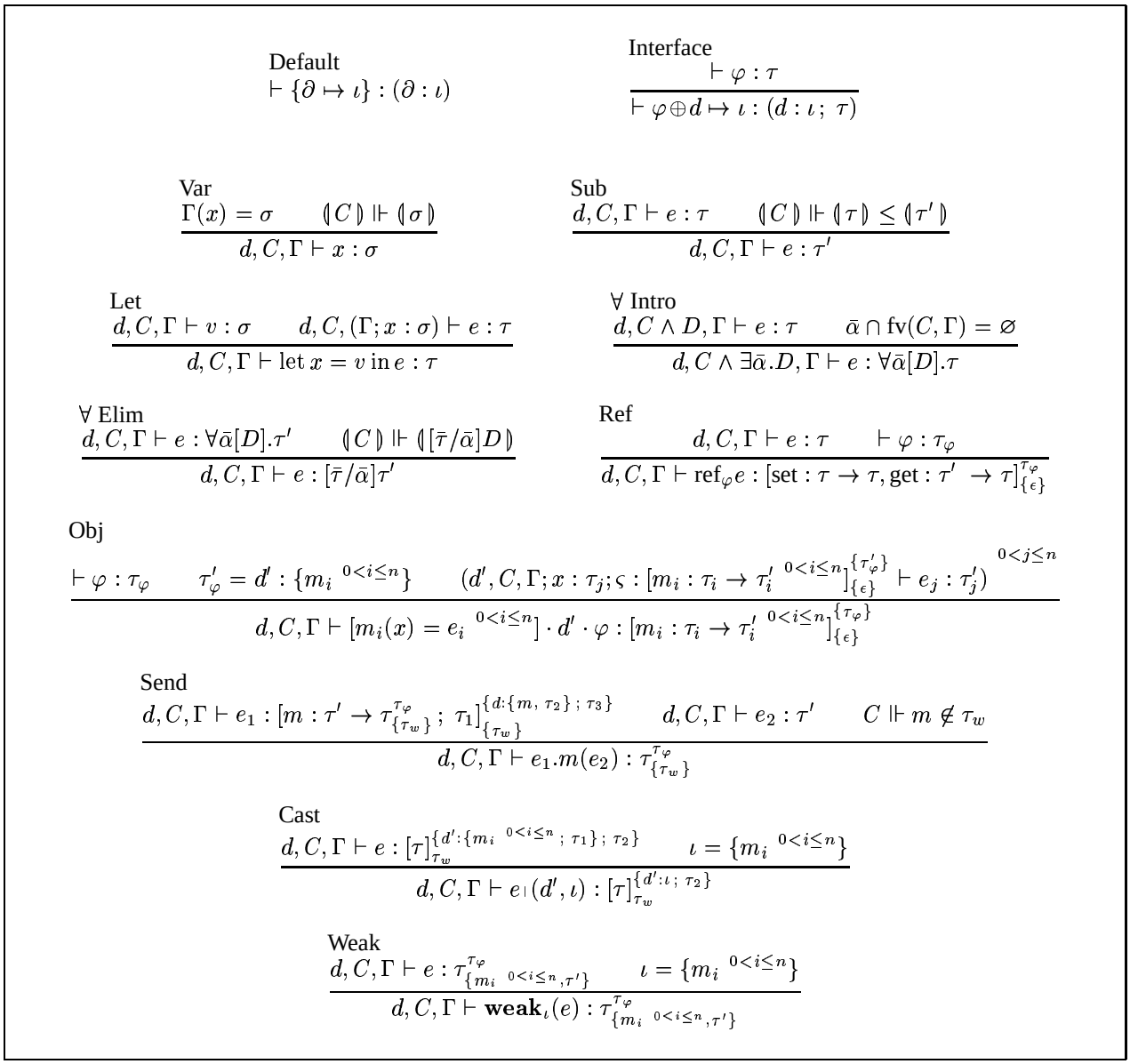


Figure 9: Direct type system for pop

We may create a version of an object equivalent to o that is `protected` with respect to the encoding of package p, as follows:

$$o_c \triangleq (\text{fctry}_{c_1}.\text{new}()) \upharpoonright (\partial, \varnothing)$$

Let o_2 be defined as follows:

$$\begin{aligned} o_2 \triangleq\ & \text{let } r_a = \text{ref}_{\{\partial \mapsto \{\text{set,get}\}\}} o_a \text{ in} \\ & \text{let } r_b = \text{ref}_{\{c_1 \mapsto \{\text{set,get}\}\}} o_b \text{ in} \\ & \text{let } r_c = \text{ref}_{\{c_1 \mapsto \{\text{set,get}\}, p \mapsto \{\text{set,get}\}\}} o_c \text{ in} \\ & \quad [\, m(x) = \varsigma.b().\text{get}(), \\ & \qquad a(x) = r_a, \\ & \qquad b(x) = r_b, \\ & \qquad c(x) = r_c \,] \cdot c_2 \cdot \varphi_2 \end{aligned}$$

Then fctry_{c_2} is encoded, similarly to fctry_{c_1}, as:

$$\text{fctry}_{c_2} \triangleq [\text{new}(x) = o_2] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\}$$

Given this encoding, if an object stored in b is leaked by a non-local use of m, it is unuseable. This is the case because, even though a non-local use of m will return b, in the encoding this return value explicitly states it cannot be used outside the confines of c_2; as a result of the definition of φ_1 and casting, the avatar o_b of b in the encoding has an interface equivalent to:

$$\{c_2 \mapsto \{f, h\}, p' \mapsto \varnothing, \partial \mapsto \varnothing\}$$

While the communication-based approach accomplishes a similar strengthening of modifier security, the benefits of greater flexibility may be enjoyed via the use-based approach. For example, a `protected` reference can be safely passed outside of a package and then back in, as long as a use of it is not attempted outside the package. Also for example are the fine-grained interface specifications allowed by this approach, enabling greater modifier expressivity— e.g. publicly read-only but privately read/write instance variables.

6 Conclusion

As shown in [1], object confinement is an essential aspect of securing OO programming languages. Related work on this topic includes the *confinement types* of [21], which have been implemented as an extension to Java [3]. The mechanism is simple: classes marked **confined** must not have references escape their defining package. Most closely related are the *ownership types* of [4]. In fact, this system can be embedded in ours, albeit with a different basis for enforcement: as discussed in Sect. 2, these previous type approaches treat a communication-based mechanism, whereas ours is use-based. One of the main points of our paper is the importance of studying the use-based approach as an alternative to the communication-based approach. Furthermore, our type system is polymorphic, with inference methods readily available due to its basis in row types.

Topics for future work include an extension of the language to capture *inheritance*, an important OO feature that presents challenges for type analysis. Also, we hope to study capability *revocation*.

In summary, contributions of this work include a focus on the more expressive use-based security model, the first type-based characterization of weak capabilities, and a general mechanism for fine-grained, use-based security specifications that includes flexible domain naming, precise object interface definitions, and domain-specific interface casting. Furthermore, we have defined a static analysis that enforces the security model, with features including flexibility due to polymorphism and subtyping, declarative benefits due to readability, and ease of proof due to the use of transformational techniques.

Bibliography

[1] Anindya Banerjee and David Naumann. Representation independence, confinement and access control. In *Conference Record of POPL02: The 29TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–177, Portland, OR, January 2002.

[2] Borris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[3] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[4] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

[5] Mark Miller *et. al.* The E programming language. URL: `http://www.erights.org`.

[6] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[7] C. Hawblitzel and T. von Eicken. Type system support for dynamic revocation, 1999. In ACM SIGPLAN Workshop on Compiler Support for System Software, May 1999.

[8] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2):202–207, February 1987.

[9] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[10] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[11] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000.

[12] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.

[13] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press.

[14] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, INRIA, 1993.

[15] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[16] Jonathan Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *21st IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[17] Christian Skalka. Syntactic type soundness for HM(X). Technical report, The Johns Hopkins University, 2001.

[18] Christian Skalka. *Types for Programming Language-Based Security*. PhD thesis, The Johns Hopkins University, 2002.

[19] Christian Skalka and Scott Smith. Set types and applications. In *Workshop on Types in Programming (TIP02)*, 2002. To appear.

[20] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.

[21] Jan Vitek and Boris Bokowski. Confined types in java. *Software—Practice and Experience*, 31(6):507–532, May 2001.

Session VI

Panel

(joint with VERIFY)

The Future of Protocol Verification

Moderators

Serge Autexier
DFKI GmbH
Saarbrüken — Germany
autexier@dfki.de

Iliano Cervesato
ITT Industries, Inc.
Alexandria, VA — USA
iliano@itd.nrl.navy.mil

Heiko Mantel
DFKI GmbH
Saarbrüken — Germany
mantel@dfki.de

Panelists

Ernie Cohen
Microsoft Research, Cambridge – UK

Alan Jeffrey
DePaul University, Chicago, IL — USA

Fabio Martinelli
CNR — Italy

Fabio Massacci
University of Trento — Italy

Catherine Meadows
Naval Research Laboratory, Washington, DC — USA

David Basin
Albert-Ludwigs-Universität, Freiburg — Germany

Abstract

This panel is aimed at assessing the state of the art and exploring trends and emerging issues in computer security in general and protocol verification in particular. It brings together experts from both the security community and the verification area. Some of questions over which they will be invited to discuss their views, and maybe even to answer, include:

- What is already solved?
- What still needs improvement?
- What are the challenging open problems?
- What is the role of automated theorem proving in protocol verification?
- What else is there in computer security besides protocol verification?

A format for this panel has been chosen as to achieve an interesting, vibrant, and productive discussion.

Author Index