Proceedings

Foundations of Computer Security

Affiliated with LICS'05

Chicago, IL
June 30 - July 1, 2005

Edited by
Andrei Sabelfeld

Table of Contents

Network Security and Denial-of-Service Attacks

Invited Talk II

Security Protocols and Decidability Issues

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 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, denial-of-service attacks, trust management, 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, then with the Workshop on Formal Methods and Computer Security—FMCS— in 2000, and finally with the LICS satellite Workshop on Foundations of Computer Security—FCS—in 2002, 2003 and 2004, of bringing together formal methods and the security community. The aim of the workshop this year is to provide a forum for continued activity in this area, to bring computer security researchers in contact with the LICS community, and to give LICS attendees an opportunity to talk to experts in computer security.

FCS received 30 submissions this year. The Program Committee selected 11 of them for presentation as the outcome of the reviewing process. In addition, the program features two invited talks, by Frank Pfenning and Jan Vitek.

The contributions of many people have made the workshop a success. The Program Committee has put much effort in providing helpful reviews. Many thanks are due to Frank Pfenning and Jan Vitek, the invited speakers, for their brave decisions to concentrate on cutting-edge research in their invited talks. Phil Scott and Radha Jagadeesan, our connections to LICS, have been of much help for FCS to run smoothly. Aslan Askarov deserves special thanks for his help in compiling the proceedings. Most of all, we are thankful to the authors and the attendees who made this workshop an inspiring and fruitful event.

Andrei Sabelfeld
FCS'05 Program Chair

Program Committee

Michael Backes, IBM Zurich, Switzerland
Gilles Barthe, INRIA, France
Iliano Cervesato, Tulane University, USA
Sabrina De Capitani Di Vimercati, University of Milan, Italy
Joshua Guttman, MITRE, USA
Joe Halpern, Cornell University, USA
Naoki Kobayashi, Tohoku University, Japan
Ralf Kuesters, University of Kiel, Germany
Cathy Meadows, NRL, USA
John Mitchell, Stanford University, USA
Frank Pfenning, Carnegie-Mellon University, USA
Mark Ryan, University of Birmingham, UK
Andrei Sabelfeld (chair), Chalmers, Sweden
Vitaly Shmatikov, University of Texas at Austin, USA

Session I

Invited Talk I

Language-Based Intrusion Detection

Jan Vitek
Department of Computer Science
Purdue University
http://www.cs.purdue.edu/people/jv

Host-based intrusion detection systems attempt to identify attacks by discovering program behaviors that deviate from expected patterns. While the idea of performing behavior validation on-the-fly and terminating errant tasks as soon as a violation is detected is appealing, this presents numerous practical and theoretical challenges. In this talk we focus on automated intrusion detection techniques, i.e. techniques which do not require human intervention. Of particular interest are techniques that rely on, or leverage, programming language semantics to find novel ways of detecting attacks. We will review the main attack models, describe the state of the art in host-based intrusion detection techniques and conclude with a list of challenges for the research community.

Session II

Language-Based Security

Dynamic Updating of Information-Flow Policies

Michael Hicks* Stephen Tse‡ Boniface Hicks† Steve Zdancewic‡
* University of Maryland † Pennsylvania State University ‡ University of Pennsylvania

Abstract

Applications that manipulate sensitive information should ensure *end-to-end* security by satisfying two properties: *sound execution* and some form of *noninterference*. By the former, we mean the program should always perform actions in keeping with its current policy, and by the latter we mean that these actions should never cause high-security information to be visible to a low-security observer. Over the last decade, security-typed languages have been developed that exhibit these properties, increasingly improving so as to model important features of real programs. No current security-typed language, however, permits general changes to security policies in use by running programs. This paper presents a simple information flow type system for that allows for dynamic security policy updates while ensuring sound execution and a relaxed form of noninterference we term *noninterference between updates*. We see this work as an important step toward using language-based techniques to ensure end-to-end security for realistic applications.

1 Introduction

Increasingly, personal and business information is being made available via networked infrastructures, so the need to protect the confidentiality of that information is becoming more urgent. A typical approach to enforcing data confidentiality is via access control. Unfortunately, access control only governs the release of information, not its propagation. Once a principal (e.g., a user, process, party, etc.) legally reads some data, he can freely share it, whether purposefully or inadvertently, despite the possible wishes of its owner. Instead, we would prefer applications to enforce *end-to-end* security by governing *information flow*: a principal should not, through error or malice, be permitted to transmit confidential information to an unauthorized party.

An information flow control system typically aims to enforce two properties: *noninterference* and *sound execution*. Given a principal hierarchy that defines the relative security levels of various principals, a program satisfies noninterference when it ensures that high security data is never visible, whether directly or indirectly, to low security observers. A program satisfies sound execution if it does not generate errors at run time. A typical way to satisfy these properties is to use a *security-typed language* [14] wherein the standard types on program variables include annotations to specify which principals are allowed to read. If a program type checks under a principal hierarchy, then, it is guaranteed that the program is noninterfering and sound with respect to the hierarchy. Security-typed languages are appealing because these properties are proven in advance of actual execution.

Typical security-typed languages assume that the principal hierarchy remains fixed during program execution. For long-running programs, such an assumption is unrealistic, as policies often change over time, e.g., to perform revocations [6, 2]. On the other hand, simply allowing the principal hierarchy to change at runtime could violate both the soundness and noninterference properties of running programs.

This paper presents a new security-typed language that allows dynamic updating of information-flow policies, particularly the delegation relations in the principal hierarchy. Section 2 defines a typical security-typed source language $\lambda^{\Pi}_{\leq}$, and then Section 4 defines λ^{Π}_{tag} as an extension of $\lambda^{\Pi}_{\leq}$ with *tags* and the ability to accommodate updates to the principal hierarchy. We prove that $\lambda^{\Pi}_{\leq}$ programs can be compiled to λ^{Π}_{tag} programs automatically, that these programs are sound, and that they respect a flavor of noninterference we dub *noninterference between*

$$
\begin{array}{llllllll}
\mathtt{p} & ::= & \mathtt{X} \mid \mathtt{p},\mathtt{p} & \ell & ::= & \mathtt{p}\colon\mathtt{p} \mid \ell,\ell & \mathcal{E} & ::= \quad \cdot \mid \mathcal{E}\,\mathtt{m} \mid \mathtt{v}\,\mathcal{E} \mid \mathtt{if}\,\mathcal{E}\,\mathtt{m}\,\mathtt{m} \\
\Pi & ::= & \cdot \mid \Pi, \mathtt{p} \leq \mathtt{p} & \mathtt{u} & ::= & \mathtt{bool}_{\ell} \mid \mathtt{u} \rightarrow \mathtt{u} & & \\
\mathtt{m} & ::= & \mathtt{true}_{\ell} \mid \mathtt{false}_{\ell} \mid \mathtt{x} \mid \lambda \mathtt{x}\colon\mathtt{u}.\,\mathtt{m} \mid \mathtt{m}\,\mathtt{m} \mid \mathtt{if}\,\mathtt{m}\,\mathtt{m}\,\mathtt{m} \mid \mathtt{if}\ (\mathtt{p} \leq \mathtt{p})\ \mathtt{m}\,\mathtt{m} & & & & &
\end{array}
$$

Figure 1: Syntax of $\lambda^{\Pi}_{\leq}$: principals $\mathtt{p}$, labels ℓ, permission Π, types $\mathtt{u}$, terms $\mathtt{m}$, and holes $\mathcal{E}$.

updates. To our knowledge, ours is the first system to safely permit general updates to the principal hierarchy, including revocations, in security-typed languages. Our discussion of the meaning of noninterference in the presence of revocation, and the definition of the term *noninterference between updates*, is also new. We believe our approach is an important step to making security-typed languages expressive enough to be used in real systems.

2 A Simple Security-Typed Language, $\lambda^{\Pi}_{\leq}$

To make our discussion of policy updates more concrete, we introduce a calculus $\lambda^{\Pi}_{\leq}$, a formalization of the *decentralized label model* [10] (DLM) based on the simply-typed lambda calculus. We present a discussion on policy updates, their challenges, and our solution to making them sound in the following two sections.

Figure 1 presents the syntax of $\lambda^{\Pi}_{\leq}$. Security policies specify confidentiality policies, defining which principals are allowed to read which data. Policies are specified in two parts. First, types and simple values are annotated with *labels* ℓ that consist of one or more pairs $(\mathtt{p}_1 : \mathtt{p}_2)$, where $\mathtt{p}_1$ is the policy owner specifying $\mathtt{p}_2$ as the reader. Principals $\mathtt{p}$ can be either literals $\mathtt{X}$ or *principal sets* $(\mathtt{p}_1, \ldots, \mathtt{p}_n)$. A label with multiple pairs, written (ℓ_1, ℓ_2) can be used to specify more restrictive policies: a potential reader must satisfy all of the label restrictions. The second part of a DLM security policy is the principal hierarchy, or a permission context, Π is represented as a list of delegations between principals $\mathtt{p}_1 \leq \mathtt{p}_2$.

Terms $\mathtt{m}$ and types $\mathtt{u}$ are largely standard. We write evaluation as $\Pi \vdash \mathtt{m}_1 \longrightarrow \mathtt{m}_2$, which states that $\mathtt{m}_1$ evaluates in a single step to become $\mathtt{m}_2$ under runtime principal hierarchy Π. The principal hierarchy can be accessed dynamically using the run-time test of principal delegation $\mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{e}_1\ \mathtt{e}_2$ [17]:

$$
\frac{\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2}{\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\ \mathtt{m}_2 \longrightarrow \mathtt{m}_1} \qquad \frac{\Pi \nvdash \mathtt{p}_1 \leq \mathtt{p}_2}{\Pi \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\ \mathtt{m}_2 \longrightarrow \mathtt{m}_2}
$$

The typing judgment has the form $\Pi; \Gamma \vdash \mathtt{m} : \mathtt{t}$, where Γ tracks the types of bound variables as usual, and permission context Π statically tracks knowledge of the principal hierarchy. Most rules are standard [17], as shown in Figure 2. The judgment $\mathtt{lab(u)} = \ell$ returns the label of the type:

$$
\mathtt{lab}(\mathtt{bool}_{\ell}) = \ell \qquad \frac{\mathtt{lab}(\mathtt{u}_2) = \ell}{\mathtt{lab}(\mathtt{u}_1 \rightarrow \mathtt{u}_2) = \ell}
$$

There are two additional typing rules. The first one type-checks the run-time test of principal delegation. If the principal delegation test succeeds, the first branch can statically assume that $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$ holds inside by adding[1] $\mathtt{p}_1 \leq \mathtt{p}_2$ to the permission context Π.

$$
\frac{\Pi, \mathtt{p}_1 \leq \mathtt{p}_2; \Gamma \vdash \mathtt{m}_1 : \mathtt{u} \qquad \Pi; \Gamma \vdash \mathtt{m}_2 : \mathtt{u}}{\Pi; \Gamma \vdash \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ \mathtt{m}_1\ \mathtt{m}_2 : \mathtt{u}} \qquad \frac{\Pi; \Gamma \vdash \mathtt{m} : \mathtt{u}_1 \qquad \Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2}{\Pi; \Gamma \vdash \mathtt{m} : \mathtt{u}_2}
$$

[1]The second branch *cannot* assume $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$; hence, there is no addition of constraints to the context for the second branch in the typing rule. Adding negative constraints ($\mathtt{p}_1 \not\leq \mathtt{p}_2$) to the context is unnecessary, because subtyping can be decided with positive constraints.

$$\Pi; \Gamma, \mathtt{x}{:}\mathtt{u} \vdash \mathtt{x} : \mathtt{u}$$

$$\frac{\Pi; \Gamma \vdash \mathtt{m}_1 : \mathtt{bool}_\ell \qquad \Pi; \Gamma \vdash \mathtt{m}_2 : \mathtt{u} \qquad \Pi; \Gamma \vdash \mathtt{m}_3 : \mathtt{u} \qquad \mathtt{lab(u)} = \ell}{\Pi; \Gamma \vdash \mathtt{if}\ \mathtt{m}_1\ \mathtt{m}_2\ \mathtt{m}_3 : \mathtt{u}}$$

$$\Pi; \Gamma \vdash \mathtt{true}_\ell : \mathtt{bool}_\ell$$

$$\frac{\Pi; \Gamma, \mathtt{x}{:}\mathtt{u}_1 \vdash \mathtt{m} : \mathtt{u}_2}{\Pi; \Gamma \vdash \lambda \mathtt{x}{:}\mathtt{u}_1.\ \mathtt{m} : \mathtt{u}_1 \rightarrow \mathtt{u}_2}$$

$$\Pi; \Gamma \vdash \mathtt{false}_\ell : \mathtt{bool}_\ell$$

$$\frac{\Pi; \Gamma \vdash \mathtt{m}_1 : \mathtt{u}_1 \rightarrow \mathtt{u}_2 \qquad \Pi; \Gamma \vdash \mathtt{m}_2 : \mathtt{u}_1}{\Pi; \Gamma \vdash \mathtt{m}_1\ \mathtt{m}_2 : \mathtt{u}_2}$$

Figure 2: Typing rules of $\lambda^{\Pi}_{\leq}$: $\Pi; \Gamma \vdash \mathtt{m} : \mathtt{t}$ (under hierarchy Π and context Γ, the term $\mathtt{m}$ has type $\mathtt{t}$).

The second one is the subsumption rule that allows the flexibility of implicitly appealing to principal delegations in the permission context during typing. There exist straightforward and efficient algorithms [7, 17] for context subtyping $\Pi_1 \leq \Pi_2$ (meaning Π_1 is more permissive than Π_2), label subtyping $\Pi \vdash \ell_1 \sqsubseteq \ell_2$ (meaning ℓ_1 is less restrictive than ℓ_2 under Π), principal subtyping $\Pi \vdash \mathtt{p}_1 \leq \mathtt{p}_2$ (meaning principal $\mathtt{p}_1$ is delegating to $\mathtt{p}_2$ under Π), and type subtyping $\Pi \vdash \mathtt{u}_1 \preceq \mathtt{u}_2$ (meaning $\mathtt{u}_1$ is a subtype of $\mathtt{u}_2$ under Π); we leave their formal specification to our companion technical report.

We have proved the desired properties of sound execution and noninterference for $\lambda^{\Pi}_{\leq}$ by a sound translation to a target language, which is to be described in Section 4.

3 The Meaning of Policy Updates

Now we consider the means and the meaning of security policy updates in $\lambda^{\Pi}_{\leq}$ (and similar languages). An information-flow security policy can change in two ways. First, the label on a particular piece of data might be altered, thereby making access to it more restricted or less restricted. The former case is permitted automatically by the subsumption rule: it is always safe to treat a piece of data more restrictively. The latter case is potentially dangerous, as the relabeling might expose sensitive information, so it is typically allowed only by an explicit *declassify* operation. A second way of changing the information-flow policy is to alter the principal hierarchy, which in turn alters the relative ordering between labels. Here again there are two kinds of changes: one can add a (directed) edge between two principals (e.g., the delegation $\mathtt{p}_1 \leq \mathtt{p}_2$), which corresponds to increasing the privileges of an observer. This kind of policy change is like a global form of declassification. One can also remove edges, corresponding to revocation, strengthening the security policy and decreasing the set of permissible information flows.

In this paper, we address the second style of policy update in which the principal hierarchy can change. To determine how and when the hierarchy should be permitted to change, we must consider the impact of policy updates on a program's security properties: sound execution and noninterference.

3.1 Models of updating

It is easy to see that naively allowing arbitrary policy updates could make evaluation *unsound*: the program could act in a way not consistent with its current policy. As an example, consider this simple program (extending the syntax presented earlier with `let` and integers):

$$\mathtt{let}\ \mathtt{x} : \mathtt{int}_{\mathtt{p}_2:} \quad = \quad (\mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ 1_{\mathtt{p}_1:}\ 3_{\mathtt{p}_2:}) \qquad \mathtt{in} \qquad \mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ \mathtt{x}\ 2_{\mathtt{p}_3:}$$

If we evaluate this program under $\Pi = \mathtt{p}_1 \leq \mathtt{p}_2$, after one step the `if` branch succeeds yielding

$$\texttt{let x} : \texttt{int}_{\mathtt{p}_2:} \quad = \quad 1_{\mathtt{p}_1:} \qquad \texttt{in} \qquad \texttt{if } (\mathtt{p}_2 \leq \mathtt{p}_3)\ \texttt{x}\ 2_{\mathtt{p}_3:} \tag{Ex 1}$$

Now say we wish to change the principal hierarchy to be $\Pi' = \mathtt{p}_2 \leq \mathtt{p}_3$. If we allow this update to occur, then the program's next evaluation step will be unsound. It will allow the data $1_{\mathtt{p}_1:}$ to flow to variable `x`, whose label $\mathtt{p}_2$: is not equal or higher under Π'. Clearly, this update should not be permitted.

Noninterference is more subtle because it is a global property: it ratifies all information flows that might occur during a program's *entire* evaluation relative to a single, fixed principal hierarchy. When the hierarchy can change, this definition no longer makes sense. One alternative is *noninterference between updates*, meaning that when a policy update occurs, the history of how some data received a certain label is forgotten, and the question of noninterference is reconsidered for the current program at the current policy. As motivation for this definition, imagine some data (a file, say) labeled as $\mathtt{p}_1$: under $\Pi'' = \mathtt{p}_1 \leq \mathtt{p}_2,\ \mathtt{p}_1 \leq \mathtt{p}_3$, meaning principals $\mathtt{p}_2$ and $\mathtt{p}_3$ are allowed to read it. If principal $\mathtt{p}_3$ is fired and $\mathtt{p}_4$ is hired, we might like to change the principal hierarchy to be $\mathtt{p}_1 \leq \mathtt{p}_2, \mathtt{p}_1 \leq \mathtt{p}_4$; i.e. to revoke the assertion $\mathtt{p}_1 \leq \mathtt{p}_3$ and add the assertion $\mathtt{p}_1 \leq \mathtt{p}_4$. From the point of view of the file labeled $\mathtt{p}_1$: and the original hierarchy Π'', the changed hierarchy both rejects flows previously allowed ($\mathtt{p}_3$ can no longer read the file) and permits flows not previously admitted ($\mathtt{p}_4$ can read the file, but could not before). Thus, at least some portion of the file's information flow history must be forgotten to permit this intuitively reasonable policy change.

On the other hand, noninterference between updates can permit unintuitive, perhaps unintended flows. For example, say the program (Ex 1) takes an additional step under Π yielding

$$\texttt{if } (\mathtt{p}_2 \leq \mathtt{p}_3)\ 1_{\mathtt{p}_1:}\ 2_{\mathtt{p}_3:}$$

Under the initial principal hierarchy Π, the program would terminate with result $2_{\mathtt{p}_3:}$. However, if we were to change the hierarchy to $\Pi''' = \mathtt{p}_1 \leq \mathtt{p}_2, \mathtt{p}_2 \leq \mathtt{p}_3$, then the `if` branch would be taken, and we would terminate with result $1_{\mathtt{p}_1:}$. The evaluation is sound and noninterfering under Π''' from the point of the change to termination. However, from the point of view of the entire program evaluation, the observed flow would have been disallowed under Π, and thus violates noninterference when considered relative to Π. Moreover, a program that satisfies noninterference between updates says nothing about the security ramifications of updates themselves. For example, one risk is that if an attacker can observe when a policy update occurs and what it consists of, he may be able to deduce the values of private data in the program.

An ideal security property would both permit policy updates to selectively "forget the past" and also reason about some flows across updates. It is an open question as to what information flow systems should enforce even when policy updates are disallowed—noninterference, though commonly supported, is too restrictive in practice. As a first step, for this paper, we ensure that program execution is sound and respects noninterference between updates, recognizing and expecting that a better property is needed. We plan to investigate stronger, adequately expressive security properties in future work.

3.2 Overview of approach

At first glance, defining principal hierarchy updates for $\lambda^{\Pi}_{\leq}$ that ensure soundness and noninterference between updates may seem straightforward. In particular, we can show noninterference between updates by proving the standard notion of noninterference: the type system is parameterized by a fixed principal hierarchy that is enforced as usual, since it implies that as long as the policy does not change, the program is noninterfering.

Proving soundness would seem equally simple: to update the principal hierarchy Π to Π' while running program `m` requires that we simply type check `m` under Π': if type checking succeeds then we permit the update. While this

approach is sound (by definition), it is overly restrictive. Consider our example program again. Say the program evaluates under Π and becomes as in Ex 1. Then say we wish to change the hierarchy to be Π'. This program will not type check since the expression $1_{\mathtt{p}_1:}$ cannot be given type $\mathtt{int}_{\mathtt{p}_2:}$ under Π'. But conceptually it should be legal, because x (which we have substituted for here with $1_{\mathtt{p}_1:}$) should be treated as having type $\mathtt{int}_{\mathtt{p}_2:}$, as defined in the original program. This fact is not revealed, however, in the run-time representation of the current state. That is, an important fact of the past (changing the type of $1_{\mathtt{p}_1:}$ to $\mathtt{int}_{\mathtt{p}_2:}$) has been forgotten.

We can solve this problem by moving away from the view of *subtyping as subset* to the view of *subtyping as coercion* for evaluation. Rather than viewing data $1_{\mathtt{p}_1:}$ as having type $\mathtt{int}_{\mathtt{p}_2:}$ under Π, we say that we can *coerce* $1_{\mathtt{p}_1:}$ to a value that has type $\mathtt{int}_{\mathtt{p}_2:}$; that is, we can coerce it to $1_{\mathtt{p}_2:}$. To do this, we extend $\lambda^{\Pi}_{\leq}$ with *permission tags* that act as coercion functions. In particular, the expression $[\ell \sqsubseteq \ell']1_\ell$ will evaluate to $1_{\ell'}$. With this change, our original program becomes

$$\mathtt{let\ x} : \mathtt{int}_{\mathtt{p}_2:} \quad = \quad \mathtt{if}\ (\mathtt{p}_1 \leq \mathtt{p}_2)\ ([\mathtt{p}_1{:} \sqsubseteq\ \mathtt{p}_2{:}]1_{\mathtt{p}_1:})\ 3_{\mathtt{p}_2:} \qquad \mathtt{in} \qquad \mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ ([\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]\mathtt{x})\ \ 2_{\mathtt{p}_3:}$$

That is, the uses of subsumption are made explicit as tags. Then the program will evaluate under Π to become

$$\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_3)\ ([\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]1_{\mathtt{p}_2:})\ 2_{\mathtt{p}_3:}$$

Now we can see that changing to Π' will be legal, as 1 has a label that can be properly typed in the new policy. At the same time, we still prevent illegal updates to the policy. In the more general case that $1_{\mathtt{p}_1:}$ were some expression $\mathtt{m}_{\mathtt{p}_1:}$, it would be unsound for the policy to change until m is a base value. Thus, while m is being evaluated (i.e., in the context of the if expression), it is guarded with the tag $[\mathtt{p}_2{:} \sqsubseteq\ \mathtt{p}_3{:}]$. An update that violated this constraint would not be allowed (as desired).

In addition to providing a more flexible coercion semantics, it turns out that permission tags can also lead to a more efficient implementation. In particular, rather than having to type check the entire program body at each proposed change in policy, we only need to look at the tags, which succinctly capture how the current policy is being used. Section 4 presents a dynamic traversal that discovers these tags at update points without having to consider function bodies. We conjecture that with only a little more work, we can adjust the evaluation semantics to keep track of the current "tag context". This would allow us to replace the traversal with a simple check.

3.3 Example

To show how these issues might arise in practice, we conclude this section with an example. Figure 3 shows a class for accessing the records of a company database, written in a Java-like syntax. This class defines two run-time principals mgr, which is a division manager, and div, which represents a division of company employees.[2] Lines 5 through 8 define some utility functions getting query inputs from the system user, processing them, creating summaries, and displaying information to the user. The policies on these methods establish that queries and the resulting processed data are owned by the mgr principal and readable by all principals in the group div, but that the results of auditing a query are only readable by mgr. These policies are explicit in the program: for example, the label on line 5 indicates that the result of get_query is owned by the principal mgr and readable by (principals in) the group div; similarly the audit method takes data readable by div and returns data only readable mgr (owners are implicitly considered to be readers in our model).

The method access_records is parameterized by a principal emp (employee), which is the current user of the database system. Line 15 dynamically checks that emp is a member of the division div, whose data is stored encapsulated in this database object. This line results in a runtime check of the principal hierarchy and succeeds only if div $\leq$ emp is true at the time when the check is made. Assuming that check succeeds, employee queries are received, processed, and displayed to the user until the user quits. In this scenario, the program also audits the

[2] Run-time principals represent principals as run-time entities, and could readily be added to our system [17].

```
01.       class Database {
02.         principal div;     /* division group */
03.         principal mgr;     /* manager for the division */
04.
05.         Query{mgr:div} get_query() {...}
06.         Data{mgr:div}  process_query(Query{mgr:div} q) {...}
07.         Data{mgr:}     audit(Data{mgr:div} d) {...}
08.         void           display(principal p, Data{mgr:p} {...}
09.
10.         void access_records(principal emp) {
11.           Query{mgr:div} query;
12.           Data{mgr:emp}  result;
13.           Data{mgr:}     summary;
14.
15.           if (div < emp) {  /* employee is a member of the division */
16.             while (true) {
17.               query = get_query();
18.               if (query == Quit) break;
19.               result  = process_query(query);
20.               summary = audit(result);
21.               display(emp, result);
22.
23.               if (mgr < emp) {  /* employee is a manager */
24.                 display(emp, summary);
25.               }
26.               ...  /* log audit information */
27.             }
28.           } else { abort(); }
29.         }
30.       }
```

Figure 3: Information-flow in a database system with principal delegations.

employee queries, perhaps to generate some statistics useful for making management decisions. The results of the audit process are readable only by managers (i.e. those principals p for which mgr $\leq$ p). For convenience, if the user of the system *is* a manager, the results of the audit are displayed immediately—the dynamic check on line 23 ensures that only managers receive this sensitive data. Presumably the program would also log the auditing information for later inspection by a manager; in this case, the current user is not able to see that data.

The code makes an important assumption: though it checks div $\leq$ emp only once, it assumes that this relationship holds for the entire execution of the `while` loop. A problem arises if this relationship is revoked while the loop executes, say if the employee is fired or just moved to a different division. In this case, an employee who no longer belonged to a particular division would still have access to its files. Even worse, if the employee were made a manager (i.e., introducing mgr $\leq$ emp into the principal hierarchy) in a new division, he would suddenly have privileges not allowed under either policy—he could read files belonging to his original division. These scenarios reveal how policy changes can violate both sound execution and our intuitive notion of noninterference.

Introducing permission tags solves these problems. In particular, to store the returned value of `process_query` into `result`, the label of the returned value must be coerced from `mgr:div` to `mgr:emp`. This will be witnessed by a coercion [mgr:div $\sqsubseteq$ mgr:emp] on `process_query(query)`, which in turn will prevent the revocation of the

$$
\begin{array}{lcl}
\mathtt{t} & ::= & \mathtt{bool}_\ell \mid \mathtt{t} \rightarrow \mathtt{t} \\
\mathtt{v} & ::= & \mathtt{true}_\ell \mid \mathtt{false}_\ell \mid \lambda[\Pi]\mathtt{x}{:}\mathtt{u}.\,\mathtt{e} \qquad \mathcal{E} ::= \cdot \mid \mathcal{E}\,\mathtt{e} \mid \mathtt{v}\,\mathcal{E} \mid \mathtt{if}\,\mathcal{E}\,\mathtt{e}\,\mathtt{e} \mid [\ell \sqsubseteq \ell]\mathcal{E} \\
\mathtt{e} & ::= & \mathtt{true}_\ell \mid \mathtt{false}_\ell \mid \mathtt{x} \mid \lambda[\Pi]\mathtt{x}{:}\mathtt{t}.\,\mathtt{e} \mid \mathtt{e}\,\mathtt{e} \mid \mathtt{if}\,\mathtt{e}\,\mathtt{e}\,\mathtt{e} \mid \mathtt{if}\,(\mathtt{p} \leq \mathtt{p})\,\mathtt{e}\,\mathtt{e} \mid [\ell \sqsubseteq \ell]\mathtt{e}
\end{array}
$$

Figure 4: Syntax of λ^{Π}_{tag}: types $\mathtt{t}$, values $\mathtt{v}$, terms $\mathtt{e}$, holes $\mathcal{E}$.

edge div $\leq$ emp from the principal hierarchy.[3]

4 A language with dynamic policy and tagging, λ^{Π}_{tag}

This section formally describes an extension to $\lambda^{\Pi}_{\leq}$, called λ^{Π}_{tag}, that permits dynamic updates to the principal hierarchy. As just described, we use permission tags of the form $[\ell_1 \sqsubseteq \ell_2]$ to prevent illegal updates of the principal hierarchy during execution. We prove that λ^{Π}_{tag} enjoys the security properties of sound execution and noninterference described earlier, even as policies change at run-time. Permission tag annotations need not burden the programmer; they can be automatically inserted by the compiler. At the end of this section, we present an automatic translation from the source calculus presented earlier to the target calculus here, based on the standard formulation of *subtyping as coercions*, and prove it sound.

The syntax of λ^{Π}_{tag} is presented in Figure 4, and closely matches the source calculus, $\lambda^{\Pi}_{\leq}$ in Section 2, except the addition of permission tags. The typing rules are the same, with one exception: the subsumption rule for subtyping is now eliminated, effectively replaced by the new typing rule for tags:

$$\frac{\Pi \vdash \ell_1 \sqsubseteq \ell_2 \qquad \Pi; \Gamma \vdash \mathtt{e} : \mathtt{bool}_{\ell_1}}{\Pi; \Gamma \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{e} : \mathtt{bool}_{\ell_2}}$$

We maintain the invariant that the current principal hierarchy Π always respects the permission tag $[\ell_1 \sqsubseteq \ell_2]$ around the term $\mathtt{e}$, as shown in the judgment $\Pi \vdash \ell_1 \sqsubseteq \ell_2$. Another invariant, which is enforced during the translation in Section 4.2, is that only boolean values are tagged. This permits the following evaluation rules:

$$\Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{true}_{\ell_1} \longrightarrow \mathtt{true}_{\ell_2} \qquad \Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{false}_{\ell_1} \longrightarrow \mathtt{false}_{\ell_2}$$

Functions are not directly tagged: they contain future computations in the body that, unlike booleans, cannot be coerced to work under different security policies. Our strategy is to extend function terms with the permission context, $\lambda[\Pi']\mathtt{x} : \mathtt{t}.\ \mathtt{e}$, such that the context Π' of the function body can be *summarized* to guard against illegal updates. The typing rules for functions and applications are:

$$\frac{\Pi \leq \Pi' \qquad \Pi'; \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\Pi; \Gamma \vdash \lambda[\Pi']\mathtt{x}{:}\mathtt{t}_1.\,\mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2} \qquad \frac{\Pi; \Gamma \vdash \mathtt{e}_1 : \mathtt{u}_1 \rightarrow \mathtt{u}_2 \qquad \Pi; \Gamma \vdash \mathtt{e}_2 : \mathtt{u}_1}{\Pi; \Gamma \vdash \mathtt{e}_1\,\mathtt{e}_2 : \mathtt{u}_2}$$

Given these tags, we can soundly and efficiently check if a policy update is legal during execution. We introduce *dynamic tag checking* $\Pi \vdash \mathtt{e}$, as shown in Figure 5, for ensuring that principal hierarchy Π is valid with respect to the running program $\mathtt{e}$. Any hierarchy is valid against boolean values. Section 4.1 proves that that the dynamic tag checking soundly approximates the static type checking with respect to the validity of policy updates.

At last, we formalize an update to the principal hierarchy of an evaluating program by defining a top-level evaluation relation. Under hierarchy Π, a program can either take a small evaluation step, or change to use the

[3] In a formal operational semantics, loops are implemented by expanding each iteration of the loop into to a fresh version of the original. Each fresh version would contain this tag, preventing any update that would violate it for the duration of the loop's execution.

$$\frac{\Pi \leq \Pi'}{\Pi \vdash \lambda[\Pi']\mathtt{x{:}t.\ e}} \qquad \frac{\Pi \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2}{\Pi \vdash \mathtt{e}_1\ \mathtt{e}_2} \qquad \frac{\Pi, \mathtt{p}_1 \leq \mathtt{p}_2 \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2}{\Pi \vdash \mathtt{if\ (p}_1 \leq \mathtt{p}_2\mathtt{)\ e}_1\ \mathtt{e}_2} \qquad \frac{\Pi \vdash \ell_1 \sqsubseteq \ell_2 \qquad \Pi \vdash \mathtt{e}}{\Pi \vdash [\ell_1 \sqsubseteq \ell_2]\mathtt{e}}$$

$$\Pi \vdash \mathtt{true}_\ell \qquad \Pi \vdash \mathtt{false}_\ell \qquad \frac{\Pi \vdash \mathtt{e}_1 \qquad \Pi \vdash \mathtt{e}_2 \qquad \Pi \vdash \mathtt{e}_3}{\Pi \vdash \mathtt{if\ e}_1\ \mathtt{e}_2\ \mathtt{e}_3}$$

Figure 5: Tag checking $\Pi \vdash \mathtt{e}$ (determines whether principal hierarchy Π is legal for the running program $\mathtt{e}$).

pending hierarchy Π'. The latter step is only permitted if the new hierarchy is legal with respect to the current program, that is, if dynamic tag checking $\Pi \vdash \mathtt{e}$ succeeds:

$$\frac{\Pi \vdash \mathtt{e} \longrightarrow \mathtt{e}'}{(\Pi; \mathtt{e})|\Pi' \longrightarrow (\Pi; \mathtt{e}')} \qquad \frac{\Pi' \vdash \mathtt{e}}{(\Pi; \mathtt{e})|\Pi' \longrightarrow (\Pi'; \mathtt{e})}$$

Note that dynamic tag checking is meant to approximate an implementation. That is, while our formulation requires a traversal over the active part of the program (i.e., the part without functional terms), this traversal could be avoided by statically gathering the set of tags S, S' that appear in the body $\mathtt{e}_1, \mathtt{e}_2$, respectively, of each $\mathtt{if\ (p}_1 \leq \mathtt{p}_2\mathtt{)\ e}_1\ \mathtt{e}_2$ expression, and then annotating the $\mathtt{if}$ with a *tag constraint* $(\mathtt{p}_1 \leq \mathtt{p}_2 \Rightarrow S) \cup S'$ (similar to conditional types [1]). These tag constraints can be maintained to form a *tag context* at run-time, so that dynamic tag checking merely considers the current tag context, rather than the active part of the program.

4.1 Security theorems

To show that the execution of a program written in our calculus is sound, we prove that any well-typed, closed term runs without any error. To show that the information flow satisfies end-to-end security, we prove that any well-typed low-security term is noninterfering by the high-security data. These *type safety* and the *noninterference* properties are formally stated as follows.

Here $\Downarrow$ is the top-level evaluation for the whole program, ignoring the number of policy updates, while $\longrightarrow^*$ is the transitive-closure of the non-updating evaluations. Therefore, type safety is guaranteed during the evaluation of the whole program, but noninterference is guaranteed between updates (as discussed in Section 3.1).

Theorem 1 (Security of dynamic policy updating)

1. *Type safety during execution: If* $\Pi; \cdot \vdash \mathtt{e} : \mathtt{t}$*, then* $(\Pi, \mathtt{e}) \Downarrow (\Pi', \mathtt{v})$.
2. *Noninterference between updates: If (1)* $\Pi; \mathtt{x} : \mathtt{bool}_{\ell_1} \vdash \mathtt{e} : \mathtt{bool}_{\ell_2}$*, and (2)* $\Pi; \cdot \vdash \mathtt{v}_1 : \mathtt{bool}_{\ell_1}$*, and (3)* $\Pi; \cdot \vdash \mathtt{v}_2 : \mathtt{bool}_{\ell_1}$*, and (4)* $\Pi \vdash \ell_1 \not\preceq \ell_2$*, then* $\Pi \vdash \mathtt{e}\{\mathtt{v}_1/\mathtt{x}\} \longrightarrow^* \mathtt{v}$ *iff* $\Pi \vdash \mathtt{e}\{\mathtt{v}_2/\mathtt{x}\} \longrightarrow^* \mathtt{v}$.

The proof for *type-safety* uses the standard technique of combining the progress and the preservation of a well-typed term. Some important lemmas for showing the soundness of tagging and tag checking are below. The first lemma states a well-typed *value* can also be well-typed under the empty principal hierarchy $\Pi = \cdot$, which is critical in the substitution lemma. The second states that the evaluation rule $\Pi \vdash \mathtt{if\ (p}_1 \leq \mathtt{p}_2\mathtt{)\ m}_1\ \mathtt{m}_2 \longrightarrow \mathtt{m}_1$ in Section 2 is type-preserving. The last lemma below shows that dynamic checking $\Pi \vdash \mathtt{e}$ is a sound approximation of static type checking $\Pi; \Gamma \vdash \mathtt{e} : \mathtt{t}$.

$$\left[\!\left[\frac{\Pi;\Gamma,\mathtt{x}:\mathtt{u}_1\vdash\mathtt{m}:\mathtt{u}_2}{\Pi;\Gamma\vdash\lambda\mathtt{x}:\mathtt{u}_1.\,\mathtt{m}:\mathtt{u}_1\to\mathtt{u}_2}\right]\!\right] = \lambda[\Pi]\mathtt{x}:[\![\mathtt{u}_1]\!].\,[\![\Pi;\Gamma,\mathtt{x}:\mathtt{u}_1\vdash\mathtt{m}:\mathtt{u}_2]\!]$$

$$\left[\!\left[\frac{\Pi;\Gamma\vdash\mathtt{m}:\mathtt{u}_1 \qquad \Pi\vdash\mathtt{u}_1\preceq\mathtt{u}_2}{\Pi;\Gamma\vdash\mathtt{m}:\mathtt{u}_2}\right]\!\right] = [\![\Pi\vdash\mathtt{u}_1\preceq\mathtt{u}_2]\!]\,[\![\Pi;\Gamma\vdash\mathtt{m}:\mathtt{u}_1]\!]$$

$$[\![\Pi\vdash\mathtt{bool}_{\ell_1}\preceq\mathtt{bool}_{\ell_2}]\!] = \lambda[\Pi]\mathtt{x}:\mathtt{bool}_{\ell_1}.\,[\ell_1\sqsubseteq\ell_2]\,\mathtt{x} \qquad \text{(fresh } \mathtt{x}\text{)}$$

$$\left[\!\left[\frac{\Pi\vdash\mathtt{u}_3\preceq\mathtt{u}_1 \qquad \Pi\vdash\mathtt{u}_2\preceq\mathtt{u}_4}{\Pi\vdash\mathtt{u}_1\to\mathtt{u}_2\preceq\mathtt{u}_3\to\mathtt{u}_4}\right]\!\right] = \lambda[\Pi]\mathtt{x}_1:[\![\mathtt{u}_1]\!]\to[\![\mathtt{u}_2]\!].\,\lambda[\Pi]\mathtt{x}_2:[\![\mathtt{u}_3]\!].\,[\![\Pi\vdash\mathtt{u}_2\preceq\mathtt{u}_4]\!]\,(\mathtt{x}_1\,([\![\Pi\vdash\mathtt{u}_3\preceq\mathtt{u}_1]\!]\,\mathtt{x}_2)) \qquad \text{(fresh } \mathtt{x}_1,\mathtt{x}_2\text{)}$$

Figure 6: Translating principal delegations to permission taggings.

Lemma 2 (Soundness of dynamic tag checking)

1. *If* $\Pi;\Gamma\vdash\mathtt{v}:\mathtt{t}$*, then* $\cdot;\Gamma\vdash\mathtt{v}:\mathtt{t}$.
2. *If* $\Pi,\mathtt{p}_1\leq\mathtt{p}_2;\Gamma\vdash\mathtt{e}:\mathtt{t}$ *and* $\Pi\vdash\mathtt{p}_1\leq\mathtt{p}_2$*, then* $\Pi;\Gamma\vdash\mathtt{e}:\mathtt{t}$.
3. *If* $\Pi;\cdot\vdash\mathtt{e}:\mathtt{t}$*, then* $\Pi\vdash\mathtt{e}$*. Moreover, if* $\Pi;\cdot\vdash\mathtt{e}:\mathtt{t}$ *and* $\Pi'\vdash\mathtt{e}$*, then* $\Pi';\cdot\vdash\mathtt{e}:\mathtt{t}$.

The proof for *noninterference* uses a logical relation for modeling the observable equivalence of a well-typed term with respect to an external observer, and shows that the substitutions preserve the equivalence [17]. Space precludes a formal development of the proofs here. Our companion technical report contains the complete rules of our calculus and the full proofs of both the type-safety and the noninterference properties. In addition, the type-safety property of the target language as well as the soundness of the translation in the next subsection are formally specified and mechanically verified[4] in Twelf (a logical framework).

4.2 Translation from $\lambda^{\Pi}_{\leq}$ to λ^{Π}_{tag}

Figure 6 shows the translation rules from the typing derivation of a $\lambda^{\Pi}_{\leq}$ term $\mathtt{m}$ to a typing derivation of a λ^{Π}_{tag} term $\mathtt{e}$. The main work is in the translation of the subsumption rule which takes the subtyping derivation of the source types and produces a well-typed *coercion function* in the target language. Breazu-Tannen et al. propose [3] such coercion semantics for the subtyping between types in the simply-typed lambda calculus. Our translation slightly extends the semantics for types with labels and permission tags. Our translation is sound as follows:

Theorem 3 (Soundness of permission tagging)

1. *Typing: If* $[\![\Pi;\Gamma\vdash\mathtt{m}:\mathtt{u}]\!]=\mathtt{e}$*, then* $\Pi;[\![\Gamma]\!]\vdash\mathtt{e}:[\![\mathtt{u}]\!]$.
2. *Subtyping: If* $[\![\Pi\vdash\mathtt{u}_1\preceq\mathtt{u}_2]\!]=\mathtt{e}$*, then* $\Pi;\Gamma\vdash\mathtt{e}:[\![\mathtt{u}_1]\!]\to[\![\mathtt{u}_2]\!]$.

We conjecture that the translation can be made *coherent* [3], meaning that target terms translated from different typing and subtyping derivations of the *same* source term have the same evaluation behavior. In particular, the

[4] We do not use the higher-order abstract syntax for encoding variable bindings. We have not performed the *totality check* which ensures that all proof cases have been completed — this property is verified externally by hand.

tag checking $\Pi \vdash \mathtt{e}$ performs the same checks whether we tag the function or the argument of an application, hence *coherent* in allowing the same set of legal policy updates. To achieve such coherent translation, *algorithmic subtyping* must be used, instead of *declarative subtyping* as presented in this paper. The conversions and theories between these variants of subtyping are standard [12].

4.3 Discussion

As mentioned in Section 3.1, the fact that a program is noninterfering between updates says nothing of possible information flows across updates. Indeed, in the system described in this section, if an attacker $\mathtt{p}_3$ can observe when updates occur, and what they consist of, it is possible for the timing of an update to communicate a secret value. Consider the following program:

$$\begin{array}{l}\mathtt{let\ x} = (\mathtt{if\ b}_{\mathtt{p}_1:} \quad (\lambda \mathtt{x{:}bool.\ true}_{\mathtt{p}_1:}) \quad (\lambda[\mathtt{p}_2 \leq \mathtt{p}_1]\mathtt{x{:}bool.}\ [\mathtt{p}_2 :\sqsubseteq \mathtt{p}_1 :]\mathtt{true}_{\mathtt{p}_2:})) \quad \mathtt{in}\\ \mathtt{let\ y} = (\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_1) \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in}\\ \mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}\end{array}$$

Suppose that the program begins evaluating with principal hierarchy $\Pi = \mathtt{p}_2 \leq \mathtt{p}_1$ and that an update $\Pi' = \emptyset$ becomes available just after $\mathtt{x}$ has been computed (call this program p). In the case that $\mathtt{b}$ was $\mathtt{true}_{\mathtt{p}_1:}$ then p would be

$$\begin{array}{l}\mathtt{let\ x} = \quad \lambda \mathtt{x{:}bool.\ true}_{\mathtt{p}_1:} \quad \mathtt{in}\\ \mathtt{let\ y} = (\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_1) \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in}\\ \mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}\end{array}$$

Thus, the policy update succeeds and $\mathtt{false}_{\mathtt{p}_3:}$ is returned. On the other hand, if $\mathtt{b}$ was $\mathtt{false}_{\mathtt{p}_1:}$, then p is

$$\begin{array}{l}\mathtt{let\ x} = \quad \lambda[\mathtt{p}_2 \leq \mathtt{p}_1]\mathtt{x{:}bool.}\ [\mathtt{p}_2 :\sqsubseteq \mathtt{p}_1 :]\mathtt{true}_{\mathtt{p}_2:} \quad \mathtt{in}\\ \mathtt{let\ y} = (\mathtt{if}\ (\mathtt{p}_2 \leq \mathtt{p}_1) \quad \mathtt{true}_{\mathtt{p}_3:} \quad \mathtt{false}_{\mathtt{p}_3:}) \quad \mathtt{in}\\ \mathtt{let\ z} = \ldots \mathit{use}\ \mathtt{x} \ldots \quad \mathtt{in\ y}\end{array}$$

Thus, the policy update is delayed due to the annotation $\mathtt{p}_2 \leq \mathtt{p}_1$ on the function $\mathtt{x}$ until $\mathtt{z}$ has been evaluated, meaning that $\mathtt{true}_{\mathtt{p}_1:}$ is returned. Hence, $\mathtt{p}_3$ is able to observe $\mathtt{b}_{\mathtt{p}_1}$ even though this is allowed by neither Π or Π'.

This particular example is an artifact of our dynamic tag checking algorithm, since it treats each branch of the initial `if` independently, once evaluated. A more static checking system, suggested earlier, would impose the same constraint on updates whichever function was chosen for $\mathtt{x}$, and eliminate this flow. Nonetheless, the noninterference between updates property is too weak to illuminate this issue or its proposed fix, so we plan to consider refinements in future work.

5 Related Work

Security-typed languages for enforcing information flow control are a rich area of research [14]. Security policies are expressed as labels on terms and a principal hierarchy defining delegation relationships; in most systems this hierarchy is fixed at compile-time. Jif [10] and recent formal work [17, 19] support *runtime principals*, which make it possible for the hierarchy to grow at runtime, but do not allow revocations. Our calculus is the first to address generalized, dynamic updates to the principal hierarchy.

Security-type systems are intended to provide a *noninterference* guarantee [13, 4, 5, 9], modulo certain small-bandwidth information channels permitted for performance reasons (timing and termination channels) and an explicit "escape hatch" in the form of a robust downgrading mechanism. The introduction of such downgrading into these languages opened a new chapter in discussions about the meaning of noninterference that is still ongoing [18, 11, 8, 15]. As we have described, our dynamic policy updating is complementary to declassification.

Declassification, as typically used, relabels a data value from one label to another; policy updates as considered in this paper permit the relationship between the labels to change over time. Both features are necessary in practice, and both can potentially be abused—it is possible that work on structured uses of declassification, as provided by robustness [18, 11] or intransitive-noninterference [8] may apply to policy updates as well. We believe our discussion of dynamic policy updating here provides a new avenue for understanding the meaning of noninterference policies for realistic programs.

This work was inspired by a similar system called Proteus that we developed for ensuring type-safety of dynamic software updates [16]. In Proteus, users can define named types T. When given that type $\mathsf{T} = \mathtt{t}$ (for some type $\mathtt{t}$), treating a value of type T as a $\mathtt{t}$ or vice versa requires an explicit coercion. When a program is dynamically updated to change the definition of T to be $\mathtt{t}'$, a dynamic analysis can check for these coercions in any functions not being updated (updated functions are assumed compatible with the new definition). If such a coercion is found then the update is only allowed if $\Gamma \vdash \mathtt{t}' \preceq \mathtt{t}$, where Γ is the updated type environment. This dynamic analysis is analogous to dynamic tag checking $\Pi \vdash \mathtt{e}$, which essentially ensures for the new Π that $\Pi \vdash \ell_1 \sqsubseteq \ell_2$ for all tags $[\ell_1 \sqsubseteq \ell_2]$ in $\mathtt{e}$. In Proteus, this dynamic analysis can be replaced with a simpler run-time given certain static information; we conjecture a similar result for λ^{Π}_{tag}.

Primarily in the context of public key infrastructures (PKI), the specific case of credentials revocation has been the subject of considerable study [6, 2]. This work has focused on the exploration of the fundamental tradeoff between security and cost. To simplify, on-line revocation servers effectively permit only a very small window of vulnerability for illicit use of compromised credentials, but often incur a high computation cost. Off-line systems provide a lower computational cost, but do so at the expense of longer latencies for receiving revocation notification. The issue of introducing policy revocation into a running program in a way that maintains sound execution has not been explored in the literature.

6 Conclusion

We have presented a new security-typed language that allows dynamic updating of information-flow policies, in particular the delegation relations in the principal hierarchy. Assumptions needed for sound execution can be represented within the program as *permission tags*, and a run-time tag checking mechanism can be used to prevent illegal updates to the principal hierarchy. Tags are implemented as run-time coercions that capture dynamic labeling behavior, which can prevent spurious rejection of legal policy updates. Tags are added to programs via an automatic translation from a standard source language. We are the first to formalize an information flow language that is sound yet permits dynamic revocations. Our language also satisfies *noninterference between updates*, which seems to us be a reasonable security property in the presence of updates. We hope that our work stimulates interest in making security-typed languages expressive enough to be used in real systems, where policies regularly change.

Acknowledgments We thank Jeff Foster, Peter Sewell, and the anonymous reviewers for their comments. Hicks is supported in part by NSF grant CCF-0346989 (*CAREER: Programming Languages for Reliable and Secure Low-level Systems*). Zdancewic and Tse are supported in part by NSF grant CCR-0311204 (*Dynamic Security Policies*) and NSF grant CNS-0346939 (*CAREER: Language-based Distributed System Security*).

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[2] D. Boneh, X. Ding, G. Tsudik, and M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of USENIX Security Symposium*, pages 297–308, Aug 2001.

[3] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[5] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, April 1984.

[6] Carl A. Gunter and Trevor Jim. Generalized Certificate Revocation. In *ACM Symposium on Principles of Programming Languages*, 2000.

[7] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *ACM Conference on Principles of Programming Languages (POPL)*, 1998.

[8] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, volume 3302 of *LNCS*, pages 129–145. Springer Verlag, 2004.

[9] John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[10] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, University of Cambridge, January 1999. Ph.D. thesis.

[11] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Asilomar, CA, June 2004. IEEE Computer Society Press.

[12] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] John C. Reynolds. Syntactic control of interference. In *Proc. 5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 39–46, 1978.

[14] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[15] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. of the 18th IEEE Computer Security Foundations Workshop*, 2005.

[16] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of theACM Conference on Principles of Programming Languages (POPL)*, January 2005.

[17] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*. IEEE Computer Society Press, May 2004.

[18] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[19] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proceedings of the 2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, Toulouse, France, August 2004.

Monitoring Information Flow

Gurvan Le Guernic Thomas Jensen
Université de Rennes 1 / CNRS
IRISA,Campus de Beaulieu, 35042 Rennes Cedex, France
{gleguern,jensen}@irisa.fr

Abstract

We present an information flow monitoring mechanism for sequential programs. The monitor executes a program on standard data that are tagged with labels indicating their security level. We formalize the monitoring mechanism as a big-step operational semantics that integrates a static information flow analysis to gather information flow properties of non-executed branches of the program. Using the information flow monitoring mechanism, it is then possible to partition the set of all executions in two sets. The first one contains executions which *are safe* and the other one contains executions which *may be unsafe*. Based on this information, we show that, by resetting the value of some output variables, it is possible to alter the behavior of executions belonging to the second set in order to ensure the confidentiality of secret data.

Keywords: security, noninterference, language-based security, information flow control, monitoring, dynamic analyses, semantics

1 Introduction

This paper is concerned with the monitoring (or dynamic analysis) of information flow in sequential programs in order to ensure confidentiality. The goal of confidentiality analysis is to ensure that secret data will not be revealed to unauthorized parties by the execution of a program [3, 6]. A by now standard way of formalizing safe information flow is via the notion of *noninterference* introduced by Goguen and Meseguer [9]. Following the notation of Sabelfeld and Myers [18], noninterference (w.r.t. some low-equivalence relations $=_L$ and $\approx_L$ on states and observations) can be expressed as follows:

$$\forall s_1, s_2 \in S.\ s_1 =_L s_2 \Rightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2 \tag{1}$$

This equation states that a command C is said to be *noninterfering* if and only if for any two states s_1 and s_2 that associate the same value to low (public) data (written $s_1 =_L s_2$), the executions of the command C in the initial state s_1 and s_2 are "low-equivalent" ($[\![C]\!]s_1 \approx_L [\![C]\!]s_2$). The "low-equivalent" relation characterizes the observational power of the attacker, by stating what he can distinguish. This may vary from requiring the output of low level of security to be equal for both executions, to requiring the two executions to have the same energy consumption. In the work presented in this paper, the attacker is considered to be only able to observe the low data of the initial state and of the final state.

As witnessed by the recent survey paper by Myers and Sabelfeld [18] there has been a substantial amount of research on static analysis for checking the noninterference property of programs, starting with the abstract interpretation of Mizuno and Schmidt [10] and the type based approach of Volpano, Smith and Irvine [21, 22]. Static analyses may reject a program because of *some* of its executions which might be unsafe; and thus deny

executions which are safe. The work presented in this paper attempt at preventing executions which are unsafe, while still allowing safe ones. This requires the definition of what is meant by "safe execution". An execution of a command C starting in the original state s_1 is said to be safe (or noninterfering) if and only if:

$$\forall s_2 \in S.\ s_2 =_L s_1 \Rightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2 \tag{2}$$

In order to allow such noninterfering executions, one approach could consist in combining a standard static information flow analysis with other static analyses in order to determine conditions on input that lead to noninterfering executions. The determination of such conditions is a difficult problem. For example, it would be possible to run a partial evaluation of the program followed by a standard information flow analysis. However there would be infinitely many partial evaluations to run, one for each set of low-equivalent initial states. The approach presented in this paper extends the execution mechanism with a monitor that allows detecting illicit information flows and forbids final states which contain illicit information flows. This will allow validating certain executions of programs beyond the reach of current static analyses, at the price of additional run-time overhead incurred by the monitoring.

Monitoring information flow is more complicated than *e.g.* monitoring divisions by zero, since it has to take into account not only the current state of the program but also the execution paths that were not taken during execution. For example, executions in an initial state where h is false and x is 0 of

(a) `if h then x := 1 else skip;`

and

(b) `if h then skip else skip;`

are equivalent concerning executed commands. However, if (b) is obviously a noninterfering program, the execution of (a) with the given initial state is not noninterfering. The execution of (a), with a low-equivalent initial state where h is true and x is 0, does not give the same final value for the low output x.

This leads to a monitoring mechanism which integrates a static analysis of commands which were not executed. The monitor will be defined formally as an operational semantics ($[\![\ ]\!]$) computing with tagged values. At any step of the evaluation of a program, the tags associated to any data identify a set of inputs which may have influenced the current value of the data up to this evaluation step. This monitoring mechanism is combined with a predicate (Safe) on the final state of the computation to obtain the following property for any command C:

$$\forall s_1 \in S.\ \mathrm{Safe}([\![C]\!]s_1) \Rightarrow (\forall s_2 \in S.\ s_2 =_L s_1 \Rightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2) \tag{3}$$

This states that all executions starting in a start state whose *low* (public) part is identical to the low part of the initial state of an execution satisfying Safe will be noninterfering (i.e. return the same values for low output). By comparison with static information flow analyses, we obtain information flow knowledge for a restricted set of input states, whereas static analyses infer a result valid for all executions. This implies a restriction of the potential paths taken into account; which enables the achievement of a better precision than with a standard static information flow analysis.

The paper is organized as follows. The next section presents a semantics integrating a monitor of information flow. It also gives a definition of the predicate Safe. This semantics and the predicate definition satisfy the equation (3), hence with this pair (semantics and predicate) it is possible to detect noninterfering executions. Once an information leak has been detected, the program behavior must be modified in order to prevent the leakage. Section 3 explores the idea of program behavior alteration based on information flow monitoring in order to ensure the respect of the confidentiality of secret data. It is observed that a simple analysis as the one developed in Section 2, although *sound* with regard to noninterference between secret inputs and public outputs, is not adequate to serve as a basis for behavior alteration. We then show a possible refinement of the analysis so that the information flow monitoring mechanism can "safely" direct the program's behavior alteration. Finally, the paper concludes by presenting some related works and possible future developments of the information flow monitoring approach.

2 Detecting noninterfering executions

The programming language considered in this paper is a sequential language with integer and boolean expressions and including loop, conditional and assignment statements. The grammar is given in Figure 1. `c` stands for any constant value, `op` for any binary operator or relation on values, and id for any variable identifier (or name).

$$
\begin{array}{rcl}
v & ::= & \texttt{c} \\
e & ::= & e_1 \ \texttt{op} \ e_2 \ \mid \ id \ \mid \ v \\
S & ::= & \textsf{if } e \textsf{ then } S \textsf{ else } S \textsf{ end} \\
 & \mid & \textsf{while } e \textsf{ do } S \textsf{ done} \\
 & \mid & S \,;\, S \ \mid \ \textsf{skip} \ \mid \ id := e
\end{array}
$$

Figure 1: Grammar of the language

Variables and values are tagged with labels, intended for indicating their security level. In order to simplify our exploration of the concepts exposed in this paper, the security lattice considered is constituted of only two elements ($\top$ and $\bot$ with the usual ordering $\bot \sqsubseteq \top$).

The special semantics on tagged data is defined as a "big step" evaluation semantics that defines an evaluation relation $\Downarrow$. It uses a value store to keep track of the value of variables. Similarly, a "tag store" is used to track the information flow between the inputs of the program and the current values in variables. Each input of the program receives a tag which reflects its security level ($\top$ for high (secret) input and $\bot$ for low (public) input). At any step of the execution, the set of tags associated to any variable by the "tag store" contains the tag of any input which has influenced the current value of the variable.

The forms of semantic judgments are described on top of Figure 2. The first environmental parameter is the value store (noted σ in the semantics rules); the second one is the tag store (noted ρ in the semantics rules). The evaluation of an expression returns a value and a set of tags. This set includes the tag of all input whose value influenced the value of the expression evaluated. For the evaluation of statements there is a third environmental parameter (noted T^{pc} in the semantics rules). It is a set of tags reflecting the information flowing through the "program counter". It contains the tag of any input which has influenced the control flow of the program up to this point in the evaluation. The evaluation of a statement returns a new value store and a new tag store reflecting all the previous information flows created and those generated by the evaluation of this statement.

2.1 SDIF: Static and Dynamic Information Flow analysis

The semantic rules are given in Figure 2 page 4. In order to reduce the number of rules and focus on the information flow computation mechanism, the semantics uses *op* and *c*. *op* is the function corresponding to the symbol `op`. Similarly, *c* is the value corresponding to the constant `c`.

The rule ($E_{\mathcal{S}}$-ASSIGN) updates the value of the variable "id" with the result of the evaluation of the expression e. It also updates the tags set of the variable in the resulting tags store. The new tags set is the union of T^e, which reflects the information flowing through the expression, and T^{pc}, which reflects the information flowing through the control flow of the program. The rule ($E_{\mathcal{S}}$-IF) evaluates the statement designated by the evaluation of the condition e, and updates the resulting tags store with the information flows created by the branch not evaluated using a special function Φ.

The function Φ (: $\mathbb{I}d \to \mathcal{P}(\mathbb{T}ag)) \times \mathcal{P}(\mathbb{T}ag) \times \mathbb{S} \to (\mathbb{I}d \to \mathcal{P}(\mathbb{T}ag))$ is used whenever an `if`-statement is evaluated. Its aim is to modify the tag store so that it reflects the information flow created by the fact that one branch of the statement is not executed. In the following program "`if h then x := k else skip end`",

$$(\mathbb{I}d \to \mathbb{V}alue);(\mathbb{I}d \to \mathcal{P}(\mathbb{T}ag)) \vdash_{\mathcal{S}} \mathbb{E}xpr \Downarrow \mathbb{V}alue : \mathcal{P}(\mathbb{T}ag)$$
$$(\mathbb{I}d \to \mathbb{V}alue);(\mathbb{I}d \to \mathcal{P}(\mathbb{T}ag));\mathcal{P}(\mathbb{T}ag) \vdash_{\mathcal{S}} \mathbb{S} \Downarrow (\mathbb{I}d \to \mathbb{V}alue) : (\mathbb{I}d \to \mathcal{P}(\mathbb{T}ag))$$

$$\frac{\sigma;\rho \vdash_{\mathcal{S}} e \Downarrow v : T^e \qquad \sigma;\rho;T^{pc} \cup T^e \vdash_{\mathcal{S}} S_v \Downarrow \sigma' : \rho'}{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} \mathsf{if}\ e\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}\ \mathsf{end} \Downarrow \sigma' : \Phi(\rho', T^e, S_{\neg v})} \quad (\mathrm{E}_{\mathcal{S}}\text{-IF})$$

$$\frac{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} \mathsf{if}\ e\ \mathsf{then}\ S\ ;\ \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{done}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{end} \Downarrow \sigma' : \rho'}{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} \mathsf{while}\ e\ \mathsf{do}\ S\ \mathsf{done} \Downarrow \sigma' : \rho'} \quad (\mathrm{E}_{\mathcal{S}}\text{-WHILE})$$

$$\frac{\sigma;\rho \vdash_{\mathcal{S}} e \Downarrow v : T^e}{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} id := e \Downarrow [id \mapsto v]\sigma : [id \mapsto T^{pc} \cup T^e]\rho} \quad (\mathrm{E}_{\mathcal{S}}\text{-ASSIGN})$$

$$\frac{}{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} \mathsf{skip} \Downarrow \sigma : \rho} \quad (\mathrm{E}_{\mathcal{S}}\text{-SKIP})$$

$$\frac{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} S_1 \Downarrow \sigma' : \rho' \qquad \sigma';\rho';T^{pc} \vdash_{\mathcal{S}} S_2 \Downarrow \sigma'' : \rho''}{\sigma;\rho;T^{pc} \vdash_{\mathcal{S}} S_1\ ;\ S_2 \Downarrow \sigma'' : \rho''} \quad (\mathrm{E}_{\mathcal{S}}\text{-SEQUENCE})$$

$$\frac{\sigma;\rho \vdash_{\mathcal{S}} e_1 \Downarrow v_1 : T_1 \qquad \sigma;\rho \vdash_{\mathcal{S}} e_2 \Downarrow v_2 : T_2}{\sigma;\rho \vdash_{\mathcal{S}} e_1\ \mathsf{op}\ e_2 \Downarrow op(v_1, v_2) : T_1 \cup T_2} \quad (\mathrm{E}_{\mathcal{S}}\text{-OP})$$

$$\frac{}{\sigma;\rho \vdash_{\mathcal{S}} id \Downarrow \sigma(id) : \rho(id)} \quad (\mathrm{E}_{\mathcal{S}}\text{-VAR})$$

$$\frac{}{\sigma;\rho \vdash_{\mathcal{S}} \mathsf{c} \Downarrow c : \emptyset} \quad (\mathrm{E}_{\mathcal{S}}\text{-VAL})$$

Figure 2: Semantics rules

the fact that x is different from k means that the then-branch has not been executed; and then that h is false. In this situation (where h is false), the final value of x is influenced by the initial value of h but not of k; even if k is the expression appearing on the right side of the assignment. The function Φ is built and used in order to take into account such information flows. A definition of the function Φ is given in Figure 3 using a combining function $\amalg$ ($\amalg \equiv \lambda f \lambda g \lambda x.\ (f\ x) \cup (g\ x)$). Φ adds the tags appearing in the tags set given in parameter to the tags set associated to any variable appearing on the left side of an assignment statement.

$$\begin{aligned}
\Phi(\rho, T, \text{“}S_1\ ;\ S_2\text{”}) &= \Phi(\rho, T, \text{“}S_1\text{”}) \amalg \Phi(\rho, T, \text{“}S_2\text{”}) \\
\Phi(\rho, T, \text{“if } e \text{ then } S_1 \text{ else } S_2 \text{ end”}) &= \Phi(\rho, T, \text{“}S_1\text{”}) \amalg \Phi(\rho, T, \text{“}S_2\text{”}) \\
\Phi(\rho, T, \text{“while } e \text{ do } S \text{ done”}) &= \Phi(\rho, T, \text{“}S\text{”}) \\
\Phi(\rho, T, \text{“}id := e\text{”}) &= [id \mapsto (\rho(id) \cup T)]\rho \\
\Phi(\rho, T, \text{“skip”}) &= \rho
\end{aligned}$$

Figure 3: Φ's semantics

The definition given here is a simple one. However it is sufficient to detect noninterfering executions with a reasonable level of precision. In the majority of cases, for programs manipulating more public inputs than secret

ones, the method presented in this section is more precise than flow insensitive analyses. Among those flow insensitive analyses are the standard security type systems which are wildly studied in the domain of language based security. In the following program, where `l` is a public input, `h` a secret input, `x` a public output and `tmp` a temporary variable, a type system would give a security level to `x` at least as high than the one of `h`; and then reject the program.

```
1  if ( l < 0 ) then { tmp := h } else { skip } end
2  if ( l > 0 ) then { x := tmp } else { skip } end
```

Using the semantics of Figure 2, all the executions of this program are detected as noninterfering (i.e. the tag of `x` at the end of the execution is $\bot$). The reason of this better precision lies in the fact that the monitoring mechanism gives us the best possible *low control flow* information: *low control flow* designates the control flow produced by branching statements whose condition has a low level of security.

The evaluation of a command produces a value store and a tag store. The notation $[\![C]\!]^{\mathbb{V}}_{\sigma,\rho}$ designates the output value store produced by the evaluation of the command C with input values σ and input tags ρ. $[\![C]\!]^{\mathbb{T}}_{\sigma,\rho}$ is similarly defined to be the output tag store. To summarize on those notations, the following holds:

$$\sigma; \rho; \emptyset \vdash_{\mathcal{S}} C \Downarrow [\![C]\!]^{\mathbb{V}}_{\sigma,\rho} : [\![C]\!]^{\mathbb{T}}_{\sigma,\rho}$$

Four sets of variables give a security specification of the program. H_i and L_i form a partition of the program's variables. H_i contains the variables holding a secret data in the initial sate (i.e. secret inputs) and L_i contains public inputs. Similarly, H_o and L_o form a partition of the program's variables in which publicly observable variables in the final state belong to L_o and unobservable variables in the final state belong to H_o. A tag store ρ is said "well-tagged" if it respects the following properties:

$$\forall x \in H_i.\ \rho(x) = \{\top\}$$
$$\forall x \in L_i.\ \rho(x) = \{\bot\}$$

Definition 2.1 (Safe**)**

$$\mathrm{Safe}([\![C]\!]^{\mathbb{T}}_{\sigma_1,\rho}) \equiv \forall x \in L_o.\ [\![C]\!]^{\mathbb{T}}_{\sigma_1,\rho}(x) \subseteq \{\bot\}$$

Using the semantics and definition of Safe presented, the following theorem is an instance of the schema in equation (3).

Theorem 2.1 *For any command C, value stores σ_1 and σ_2, and "well-tagged" tag store ρ, such that* $\mathrm{Safe}([\![C]\!]^{\mathbb{T}}_{\sigma_1,\rho})$ *and $[\![C]\!]^{\mathbb{V}}_{\sigma_2,\rho} \neq \bot$, if $\sigma_1 =_{L_i} \sigma_2$ then $[\![C]\!]^{\mathbb{V}}_{\sigma_1,\rho} =_{L_o} [\![C]\!]^{\mathbb{V}}_{\sigma_2,\rho}$.*

This theorem states that, for a given command, if the low outputs of an execution ϵ are all tagged with $\bot$, then for all other terminating execution if the low inputs are equal to those of ϵ, ($\sigma_1 =_{L_i} \sigma_2$), then the low outputs will be equal to those of ϵ, ($[\![C]\!]^{\mathbb{V}}_{\sigma_1,\rho} =_{L_o} [\![C]\!]^{\mathbb{V}}_{\sigma_2,\rho}$).

The theorem 2.1 is similar to the equation (3) given in introduction. In fact, as the attacker can only observe the low outputs, the equality of the low outputs ($[\![C]\!]^{\mathbb{V}}_{\sigma_1,\rho} =_{L_o} [\![C]\!]^{\mathbb{V}}_{\sigma_2,\rho}$) matches the equivalence of the final states as defined in the equation ($[\![C]\!]s_1 \approx_L [\![C]\!]s_2$). And similarly, the equality of low inputs ($\sigma_1 =_{L_i} \sigma_2$) corresponds to the low equivalence of the initial states. The only visible difference is the statement "$[\![C]\!]^{\mathbb{V}}_{\sigma_2,\rho} \neq \bot$" in the theorem 2.1. However, as the attacker is unable to observe the termination behavior of the program, this statement is implied by the definition of low-equivalence of final states used in the equation (3). Therefore, we can conclude that if all the low outputs are tagged with $\bot$ then the current execution is noninterfering, and then an attacker is unable to deduce any information about the high inputs.

To illustrate what precede, the result of the evaluation of the following program P is given in Table 1.

```
1  x := 0;
2  if l then
3    if h then { x := 1 } else { skip } end
4  else { skip } end
```

In this program, x is a low level output, l is a low level input (with tag $\bot$), and h is a high input (with tag $\top$).

$\sigma(l)$	$\sigma(h)$	$[\![P]\!]^{\mathbb{V}}_{\sigma,\rho}$(x)	$[\![P]\!]^{\mathbb{T}}_{\sigma,\rho}$(x)
True	True	1	$\top$
True	False	0	$\top$
False	True	0	$\bot$
False	False	0	$\bot$

Table 1: Results for the output x

3 Altering the program's behavior

The semantics described in the previous section enables the *detection* of a subset of noninterfering executions. The next step consists in the alteration of programs behavior in order to *enforce* the confidentiality of secret data. Our goal is to ensure that the set of all altered executions, for any program P, respects the noninterference property of Goguen and Meseguer as defined by the equation (1). This property states that any execution of the program is a noninterfering execution as defined by the equation (2). Consequently, the behavior alteration consists in:

- doing nothing for executions which are detected as noninterfering,
- modifying the output values of executions which may be interfering.

The altered execution of the program P started in the initial state s is noted $\widetilde{[\![\mathsf{P}]\!]s}$.

The predicate Safe partitions the set of executions of the program P into two sets $\mathcal{E}_{ni}$ (containing the executions for which the predicate Safe is true) and $\mathcal{E}_?$ (containing the executions for which the predicate Safe is false). From the equation (3), we know that all the executions in $\mathcal{E}_{ni}$ are noninterfering. The problem lies in the executions of $\mathcal{E}_?$ among which some are noninterfering and some are not, and so may reveal information about the secret data. The solution envisioned consists in using a default output state s^d_o. As it is possible to detect, during the execution of the program P, if the current execution belongs to $\mathcal{E}_{ni}$ or $\mathcal{E}_?$, it is possible to force the output store of all the executions belonging to $\mathcal{E}_?$ to be s^d_o. Then, for any program P and initial state s_1 the following properties hold:

$$\mathrm{Safe}([\![\mathsf{P}]\!]s_1) \Rightarrow (\ \mathrm{Safe}(\widetilde{[\![\mathsf{P}]\!]s_1})\ \wedge\ (\forall s_2 \in S.\ s_2 =_L s_1 \Rightarrow \widetilde{[\![\mathsf{P}]\!]s_1} = [\![\mathsf{P}]\!]s_1 \approx_L [\![\mathsf{P}]\!]s_2 = \widetilde{[\![\mathsf{P}]\!]s_2})\ ) \tag{4}$$

$$\neg\mathrm{Safe}([\![\mathsf{P}]\!]s_1) \Rightarrow (\ \neg\mathrm{Safe}(\widetilde{[\![\mathsf{P}]\!]s_1})\ \wedge\ \widetilde{[\![\mathsf{P}]\!]s_1} = s^d_o\ ) \tag{5}$$

If the predicate Safe *gives the same answer for any two executions started in low-equivalent states,* then the equations (4) and (5) imply that for all altered executions of any program P the following holds:

$$\forall s_1, s_2 \in S.\ s_2 =_L s_1 \Rightarrow (\ (\widetilde{[\![\mathsf{P}]\!]s_1} \approx_L \widetilde{[\![\mathsf{P}]\!]s_2})\ \vee\ \widetilde{[\![\mathsf{P}]\!]s_1} = \widetilde{[\![\mathsf{P}]\!]s_2} = s^d_o\ ) \tag{6}$$

It is then obvious that the set of all altered executions, for any program P, respects the noninterference property of Goguen and Meseguer as defined in equation (1).

The following example illustrates the ideas exposed above using a program transformation altering the final value of the output x depending on its final tag.

```
1  x := 0;
2  if h then
3    if l then { x := 1 } else { skip } end
4  else { skip } end
5  if (T in tag(x)) then { x := 2 }
```

The 4 first lines correspond to the original program in which x is a low level output, l is a low level input (with tag $\bot$), and h is a high input (with tag $\top$). The 5^{th} line is added to prevent information leakage. If, at the beginning of line 5, the tag of x contains $\top$, then x is reset to a default value (2 in this case, it could be what ever value is desired). The idea behind the 5^{th} line is that **if**, at the beginning of line 5, x may have different values for two executions having the same low inputs **then** the tag of x will be $\top$; so the test of the $5th$ line will succeed for both executions **and then** x will be reset to the same value (2 in this case) for both executions. This way, the program has been corrected in order to respect the noninterference property.

The tag, as computed by the semantics given in Section 2, at the end of line 4 (i.e. just before the information flow test) is given in Table 2 as a function of the input value of l (horizontally) and h (vertically). In this program,

σ(h) \ σ(l)	True	False
True	$\top$	$\bot$
False	$\top$	$\top$

Table 2: $[\![P]\!]^{\mathbb{T}}_{\sigma,\rho}$(x)

if l is true it is possible to deduce the value of h by looking at the value of x before line 5. If x is 1 then h is true, and if x is 0 then h is false. This is reflected by the tag of x which is $\top$ in both cases. Consequently, the value of x will be reset in both cases; those two altered executions of the program will then respect the noninterference property (i.e. the value of the output x is identical whatever the value of the high input is). Nevertheless, the statement added for correction is troublesome in a situation which was safe without it.

If l is false then x is equal to 0 whatever the value of h is. This means that those two executions respect the noninterference property before line 5. However, the tag of x is $\bot$ if h is true, and $\top$ if h is false. Both tags are correct because there is no flow from h to x and the tag reflects only a "*may* influence" relation. The problem with those tags is that, in the case where l is false, the correcting statement will change the value of x if and only if h is false. So, in the case where l is false, the value of x after the line 5 depends on the value of h. This implies that the set of all altered executions of the program does not respect the noninterference property.

3.1 A fully dynamic tag semantics

As shown in what precedes, in order for the equation (6) to holds, it is required that the predicate Safe returns the same answer for two executions started in low-equivalent states. If and only if that is the case, it is possible to secure programs based on the information flow computed dynamically. In our case, it means that the semantics must compute the same output tag stores for any two executions having the same low inputs. It is not the case for the semantics studied in Section 2.

Another semantics, whose rules can be found in Figure 4 page 8, has been developed. This semantics goes through all possible paths in order to compute adequate tags. When it encounters a branching statement it evaluates completely the branch that the condition designates (i.e. computes the new value store and tag store), and computes

$$\frac{\sigma;\rho \vdash_{\mathcal{F}} e \Downarrow v : T^e \quad \sigma;\rho;T^{pc}\cup T^e \vdash_{\mathcal{F}} S_{true} \Downarrow \sigma_{true} : \rho_{true} \quad \sigma;\rho;T^{pc}\cup T^e \vdash_{\mathcal{F}} S_{false} \Downarrow \sigma_{false} : \rho_{false}}{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} \textsf{if } e \textsf{ then } S_{true} \textsf{ else } S_{false} \textsf{ end} \Downarrow \sigma_v : \{\!|\rho_{true},\rho_{false}|\!\}_v^{T^e}} \quad (\text{E}_{\mathcal{F}}\text{-IF})$$

$$\frac{\sigma;\rho \vdash_{\mathcal{F}} e \Downarrow v : T^e \quad \sigma;\rho;T^{pc}\cup T^e \vdash_{\mathcal{F}} S \,;\, \textsf{while } e \textsf{ do } S \textsf{ done} \Downarrow \sigma' : \rho'}{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} \textsf{while } e \textsf{ do } S \textsf{ done} \Downarrow \{\!|\sigma',\sigma|\!\}_v^{\emptyset} : \{\!|\rho',\rho|\!\}_v^{T^e}} \quad (\text{E}_{\mathcal{F}}\text{-WHILE})$$

$$\frac{\sigma;\rho \vdash_{\mathcal{F}} e \Downarrow v : T^e}{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} id := e \Downarrow [id \mapsto v]\sigma : [id \mapsto T^{pc}\cup T^e]\rho} \quad (\text{E}_{\mathcal{F}}\text{-ASSIGN})$$

$$\frac{}{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} \textsf{skip} \Downarrow \sigma : \rho} \quad (\text{E}_{\mathcal{F}}\text{-SKIP})$$

$$\frac{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} S_1 \Downarrow \sigma' : \rho' \qquad \sigma';\rho';T^{pc} \vdash_{\mathcal{F}} S_2 \Downarrow \sigma'' : \rho''}{\sigma;\rho;T^{pc} \vdash_{\mathcal{F}} S_1 \,;\, S_2 \Downarrow \sigma'' : \rho''} \quad (\text{E}_{\mathcal{F}}\text{-SEQUENCE})$$

$$\frac{\sigma;\rho \vdash_{\mathcal{F}} e_1 \Downarrow v_1 : T_1 \qquad \sigma;\rho \vdash_{\mathcal{F}} e_2 \Downarrow v_2 : T_2}{\sigma;\rho \vdash_{\mathcal{F}} e_1 \textsf{ op } e_2 \Downarrow \mathit{op}(v_1,v_2) : T_1 \cup T_2} \quad (\text{E}_{\mathcal{F}}\text{-OP})$$

$$\frac{}{\sigma;\rho \vdash_{\mathcal{F}} id \Downarrow \sigma(id) : \rho(id)} \quad (\text{E}_{\mathcal{F}}\text{-VAR})$$

$$\frac{}{\sigma;\rho \vdash_{\mathcal{F}} \odot \Downarrow \odot : \emptyset} \quad (\text{E}_{\mathcal{F}}\text{-VAL})$$

$$\{\!|x,y|\!\}_v^{T^e} = \begin{cases} x \cup y & \text{if } \top \in T^e \\ x & \text{if } \top \notin T^e \text{ and } v = true \\ y & \text{if } \top \notin T^e \text{ and } v = false \end{cases}$$

Figure 4: Rules of the full-paths semantics

the new tag store returned by the evaluation of the other branch. The tag store the semantics returns in such a situation is the join of the two tag stores (one for each branch). Using this semantics, the following theorem has been proved to hold.

Theorem 3.1 *For any command C, value stores σ_1 and σ_2, and "well-tagged" tag store ρ, such that* $\mathrm{Safe}(\llbracket C \rrbracket^{\mathbb{T}}_{\sigma_1,\rho})$ *and* $\llbracket C \rrbracket^{\mathbb{V}}_{\sigma_2,\rho} \neq \bot$*, if* $\sigma_1 =_{L_i} \sigma_2$ *then* $\llbracket C \rrbracket^{\mathbb{V}}_{\sigma_1,\rho} =_{L_o} \llbracket C \rrbracket^{\mathbb{V}}_{\sigma_2,\rho}$ *and* $\llbracket C \rrbracket^{\mathbb{T}}_{\sigma_1,\rho} =_{L_o} \llbracket C \rrbracket^{\mathbb{T}}_{\sigma_2,\rho}$.

This is sufficient to be able to safely alter the behavior of programs in order to ensure the respect of the noninterference property. Nevertheless, the semantics used is highly inefficient. For any execution of a program, the semantics evaluates all paths which are accessible by any execution started in a low-equivalent initial state. Moreover, as soon as the semantics encounters a **while**-statement branching on a condition influenced by a high level input (but not if the condition depends only on public inputs), the semantics loops forever. This is quite disturbing and the reason for the current development of another semantics.

4 Related Works

The vast majority of information flow analyses are static and involve type systems [18]. In the recent years, this approach has reached a good level of maturity. Pottier and Conchon described in [16] a systematic way of producing a type system usable for checking *noninterference*. Profiting from this maturity, some "real size" languages including a security oriented type system have been developed. Among them are JFlow [11], JIF [14], and FlowCaml [19, 17]. There also exists an interpreter for FlowCaml. This interpreter dynamically type data, commands and functions which are successively evaluated. Nevertheless, it types commands the same way the static analysis does. And then, the interpreter merges the types of both branches of an **if**-statement without taking into account, when possible, the fact that one branch is executed and the other one is not.

One of the drawbacks of type systems concerns the level of approximation involved. In order to improve the precision of those static analyses, dynamic security tests have been included into some languages and taken into account in the static analyses. The JFlow language [11, 12], which is an evolution of Java, uses the *decentralized label model* of Myers and Liskov [13]. In this model, variables receive a label which describes allowed information flows among the principals of the program. Some dynamic tests of the principals hierarchy and variables labels are possible, as well as some labels modifications [26]. Zheng and Myers [27] include dynamic security labels which can be read and tested at run-time. Nevertheless, labels are not computed at run-time. Using dynamic security tests similar to the Java stack inspection, Banerjee and Naumann developed in [2] a type system guarantying noninterference for well-typed programs and taking into account the information about the calling context of method given by the dynamic tests.

Going further than *testing* dynamically labels, there has been research on dynamically *computing* labels. At the level of languages, Abadi, Lampson, and Lévy expose in [1] a dynamic analysis based on the labeled λ-calculus of Lévy. This analysis computes the dependencies between the different parts of a λ-term and its final result in order to save this result for a faster evaluation of any future equivalent λ-term. Also based on a labeled λ-calculus, Gandhe, Venkatesh, and Sanyal [8] address the information flow related issue of *need*. It has to be noticed that even some "real world" languages dispose of similar mechanisms. The language Perl includes a special mode called "Perl Taint Mode" [23]. In this mode, the *direct* information flows originating with user inputs are tracked. It is done in order to prevent the execution of "bad" commands. None of those works take into account *indirect flows* created by the non-execution of one of the branches of a statement. At the level of operating systems, Weissman [24] described at the end of the 60's a security control mechanism which dynamically computes the security level of newly created files depending on the security level of files previously opened by the current job. Following a similar approach, Woodward presents its *floating labels* method in [25]. This method deals with the problem of over-classification of data in computer systems implementing the MAC security model. The main difference between those two works and ours lies in the granularity of label application. In those models [24, 25], at any time, there is only one label for all the data manipulated. Data's "security levels" cannot evolve separately from each other. More recently, Suh, Lee, Zhang, and Devadas presented in [20] an architectural mechanism, called *dynamic information flow tracking*. Its aim is to prevent an attacker to gain control of a system by giving *spurious* inputs to a program which may be buggy but is not malicious. Their work looks at the problem of security under the aspect of integrity and does not take care of information flowing indirectly throw branching statements containing different assignments. At the level of computers themselves, Fenton [7] describes a small machine, in which storage locations have a *fixed* data mark. Those data marks are used to ensure a secure execution with regard to noninterference between private inputs and non-private outputs. However, the fixed characteristic of the data marks forbids modularity and reuse of code. As Fenton shows himself, his mechanism does not ensure confidentiality with *variable* data marks. At the same level, Brown and Knight [4] describe a machine which dynamically computes security level of data in memory words and try to ensure that there are no undesirable flows. This work does not take care of non-executed commands. As it has been shown in this paper, this is a feature which can be used to gain information about secrets in some cases. For example, Table 1 shows that it

is possible to deduce the value of h when l is true and $[\![P]\!]^{\mathbb{V}}_{\sigma,\rho}$(x) is 0; even if no assignment to l or x has been executed. With a program similar to the one used as example in page 6, their machine does not prevent the flow from h to x when l is true and h is false.

5 Conclusion

In this paper, we refine the notion of noninterference, concerning all possible executions of a program, to a notion of noninterfering execution. All possible initial states of a program are partitioned in equivalence classes. The equivalence relation is based on the value of the public inputs of the program. Two initial states are equivalent if and only if they have the same values for public inputs. An execution, started in the initial state s, is said to be noninterfering if any execution, started in an initial state belonging to the same equivalence class than s, returns the same values for the public outputs of the program.

Refining the notion of noninterference to the level of execution offers two main advantages. The first one is that it is now possible to *safely* run noninterfering executions of a program which is not noninterfering. The second benefit is a better precision in the analysis of some programs. A static information flow analysis has to take into consideration all the potential paths of all the executions of the program. Using the method presented in this paper to ensure the respect of confidentiality, only the potential paths of executions low-equivalent to the current one are taken into consideration. This feature results in a better precision towards possible execution paths. For example, in the following program, h is a secret input, l a public input, tmp a temporary variable which is not an output, and x is the only public output.

```
1  if ( (cos l)^2 < 0.1 ) then { tmp := h } else { skip } end
2  if ( (tan l) < 3 ) then { x := tmp } else { skip } end
```

It is likely that a static analysis would conclude that the program is not noninterfering because of a bad flow from h to x. However, the program *is* noninterfering. As "$(\cos x)^2 + (\sin x)^2 = 1$" and "$\tan x = \frac{\sin x}{\cos x}$", there is no l such that $(\cos l)^2 < 0.1$ and $(\tan l) < 3$. It follows that there is no execution of the program which evaluates both assignments. Consequently, there is never a flow from h to x. The mechanism proposed in this paper would allow all executions of this program. The reason is that, for any low-equivalent class of executions, there is exactly *one* possible path. And so, only the current execution path is taken into consideration when determining if a given execution is noninterfering or not.

Concerning the capacity of the attacker, this work considers an attacker which is only able to get information from the low outputs of the program at the end of the computation. Another limitation concerns termination of programs. The mechanism developed here does not prevent information leakage from the study of the termination behavior of programs (neither does it take care of timing covert channels either). The system proposed in this paper could prevent those flaws using a technique similar to the one found in [5]. In short, the authors of this paper track the security level of variables appearing in while-loop conditions and other statements influencing the termination. This is efficient but restrictive since it forbids any loop conditioned by a secret. That is the reason why those types of covert channels are not taken into consideration at first.

We propose a special semantics and a predicate on the final state of an execution which, together, are able to detect noninterfering executions. This semantics mixes dynamic mechanism and static analysis techniques. When the semantics encounters a branching statement, the branch designated by the value of the condition is evaluated and the other branch is analyzed. The aim of the analysis is to extract the information flow created by the fact that the given branch is not executed. The result of the analysis and the evaluation of the other branch are merged together to build the resulting information flows corresponding to the evaluation of the branching statement.

The next step of this work consists in altering programs behavior in order to ensure an appropriate behavior of programs towards confidentiality. However, the first semantics presented does not necessarily return the same

result about noninterference for two executions whose initial states belong to the same equivalence class. This prevents the use of this semantics for the programs behavior alteration in order to ensure confidentiality. In Section 3 we describe succinctly a first attempt at improving the semantics. The resulting semantics is proved sufficient to ensure the respect of confidentiality by all altered execution of any program. However, this semantics does not terminates for programs containing a while-statement conditioned by a "secret" data.

Future work will involve the development of a semantics having a precision enabling the insertion of dynamic tests, but having better termination properties. This semantics will use an analysis of non-executed branches based on the model of flow logic [15] in a way similar to [5] since this model seems to have a good precision. In particular, it does not require that a variable keeps the same security level in all the statements. The precision of this model will be improved by taking into account the knowledge (i.e. the value store) gathered by the semantics up to the starting point of the analysis.

Acknowledgment. Discussions with David Schmidt and Anindya Banerjee during the development of this work have been helpful; as well as their comments on this paper.

References

[1] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proc. ACM International Conf. on Functional Programming*, pages 83–91, 1996.

[2] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 155–169, 2003.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, May 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[4] J. Brown and T. F. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical Report ARIES-TM-015, MIT, Nov. 2001.

[5] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *J. Computing Languages*, 28(1):3–28, 2002.

[6] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[7] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[8] M. Gandhe, G. Venkatesh, and A. Sanyal. Labeled lambda-calculus and a generalized notion of strictness (an extended abstract). In *Proc. Asian C. S. Conf. on Algorithms, Concurrency and Knowledge*, pages 103–110, 1995.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[10] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *J. Formal Aspects of Computing*, 4(6A):727–754, 1992.

[11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. Principles of Programming Languages*, pages 228–241, 1999.

[12] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, 1999.

[13] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. Security and Privacy*, pages 186–197, 1998.

[14] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow, 2001. Soft. release. http://www.cs.cornell.edu/jif.

[15] H. R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 223–244, 2002.

[16] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conf. on Functional Programming*, pages 46–57, 2000.

[17] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[19] V. Simonet. Fine-grained information flow analysis for a λ-calculus with sum types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 223–237, 2002.

[20] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[21] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 607–621, 1997.

[22] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[23] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl, 3rd Edition*. O'Reilly, July 2000.

[24] C. Weissman. Security controls in the adept-50 timesharing system. In *Proc. AFIPS Fall Joint Computer Conf.*, volume 35, pages 119–133, 1969.

[25] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *Proc. IEEE Symp. Security and Privacy*, pages 23–31, 1987.

[26] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.

[27] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proc. Workshop Formal Aspects in Security and Trust*, 2004.

Optimized Enforcement of Security Policies *

M. Langar M. Mejri
LSFM Research Group
Computer Science Department,
Laval University, Sainte-Foy, Qc, G1K 7P4, Canada.

Abstract

Given a program P and a security policy Φ, this paper introduce an approach that can automatically produce another P' such that $P' \models \Phi$ and P and P' are "equivalent" (with respect to a precise de finition of equivalence). In reality, the program P' is the program P to which some tests are added in some critical places so that the security policy will be respected. Finally, some static analysis are performed on P', using typing system, in order to eliminate tests that can be statically verified.

Key Words: Security policy, Monitoring, Typing System.

1 Introduction

With the dazzling proliferation of Internet and distributed systems such as the world wide web, combined with the convergence of voice, data and image, the mobile code has seen an important increasing use. Basically, a mobile code is a program (ActiveX controls, Java applets, script run within the browser, etc.) that is downloaded and executed in our machine (computer, PDA, mobile phone, etc.); generally, without our explicit request. Since the sources of mobile codes could be malicious, they are generally untrusted and the responsibility of verifying their safety belong to the consumers. The machines of consumers must maintain a strict control over mobile codes so that they will never violate their requested security policy.

Despite the great progress in the computer security research field, fully secure computer systems are still a distant dream. In one hand, this is due to the subtleness and the complexity of the problem. And, in the other hand, it is due to the missing of efficient and powerful formal methods to deal with this kind of problems. As a result, many computer system in the world suffer for security flaws.

Basically the literature records two classes of techniques that deal with the verification of computer system: static analysis [1, 2, 3] that consists on checking programs before their executions and dynamic analysis [9, 10, 11] that consists on checking the programs during their executions.

Generally, these techniques complement each other since there are some properties that could not be verified dynamically and vice-versa. For instance, liveness properties (some thing good will happen) could not be ensured dynamically. Other properties that depend on some values known only at run time could not be verified statically. Readers are invited to see [4] for more details about properties that could be verified statically and the ones

*This research is supported by NSERC (the Natural Sciences and Engineering Research Council of Canada) and FQRNT (Fonds Québécois de la Recherche sur la Nature et les Technologies)

that could be verified dynamically. Since dynamic analysis may slow considerably the execution of programs and requests supplementary memory, it is generally recommended to very statically all properties that could be verified both statically and dynamically.

In this paper, we propose a formal approach that allows us to enforce security policies in an efficient way. More precisely, given a program P and a security policy Φ, we want to produce another program P' such that the following conditions hold:

- $P' \models \Phi$; meaning that the program P' respects the required security policy.
- P and P' are two "equivalent" programs; meaning that all the sequences of actions that can be performed by P' could be performed by P and all the sequences of actions that could be performed by P without violating the security policy could be also performed by P'.
- All parts of Φ that could be statically verified haven't to be taken into consideration by P'.
- The parts of Φ that could not be verified statically have to be taken into consideration by P' in an efficient way. In other words, we will produce P' from P by inserting "in the right place" only the necessary tests.

The rest of this paper is structured as follows : Section 2 defines the logic used to specify security policies. Section 3 presents the syntax and the semantics of our process algebra. Section 4 show how we can easily ensure that a program can never violate its security policy. Section 5 defines a typing system that statically optimizes the tests requested by security policies. Section 6 illustrates our technique by some examples. Section 7 deals with the equivalence of a program and its optimized version. Finally, in Section 8 some concluding remarks on this work and future research are ultimately sketched as a conclusion.

2 Logic

In this section, we define a logic for the specification of security properties that we can enforce dynamically. Security properties can be divided in two classes: safety properties and liveness properties. The first class expresses the fact that "bad things should never happen during the execution of a program", and the second class expresses the fact that "good things must happen". In a dynamic context, there is no way to enforce liveness properties [5]. So, in this study we focus only on the class of safety properties. Such properties can be expressed by regular expressions (e.g. Security Automata [5]) or by logic (e.g. LTL [6]). The logic used in this paper will be denoted by L_φ and it is inspired from Kleene algebras [7] and regular expressions. Basically, it allows us to specify properties which can be checked on a trace-based model, and properties related to infinite behavior (e.g. a server shouldn't send the pass-word of users). The choice of this logic is motivated by its syntax that is close to the one chosen for processes and this similarity is helpful to simplify the static and dynamic analysis steps.

2.1 syntax

The syntax of the logic L_φ is given by the BNF grammar shown by Tab. 1.

In this syntax, p is a proposition related to atomic actions. It could be an effect (an atomic action) under a condition such as $send(x) : x \geq 0$, $read(x) : x \leq 0$, etc., or a boolean expression such $x = 5$, $x > 3$, etc. Also, a proposition could be effect without condition, in this case we write $f : tt$. The formula $\phi_1.\phi_2$ means that the program must respect ϕ_1 and then ϕ_2. And, the formula $\phi_1^*\phi_2$ means that the program must respect ϕ_1 repeatedly and as soon as ϕ_1 is violated this program must respect ϕ_2 (It is the continuous version of the Kleene operator* [7]).

Table 1: Syntax of L_φ.

$\phi \ ::= \ p \mid \phi_1.\phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1^*\phi_2$
$p \ ::= b \mid f : b$
$f \ ::= read(x_1, \ldots, x_{n_1}) \mid write(x_1, \ldots, x_{n_2}) \mid send(x_1, \ldots, x_{n_3}) \mid \ldots$

Note that usual shortcuts such that $\wedge$, $\rightarrow$, $\leftrightarrow$, tt and ff, $F(\Phi)$ (eventually Φ) and $G(\Phi)$ (always Φ) can be used with there usual meaning shown hereafter:

$$\begin{array}{rclcrcl} tt & \equiv & \neg\Phi \vee \Phi & & ff & \equiv & \neg tt \\ \Phi_1 \wedge \Phi_2 & \equiv & \neg(\neg\Phi_1 \vee \neg\Phi_2) & & \Phi_1 \rightarrow \Phi_2 & \equiv & \neg\Phi_1 \vee \Phi_2 \\ F(\Phi) & \equiv & \neg G(\neg\Phi) & & G(\Phi) & \equiv & \Phi^* ff \\ \Phi_1 \leftrightarrow \Phi_2 & \equiv & \Phi_1 \rightarrow \Phi_2 \wedge \Phi_2 \rightarrow \Phi_1 & & & & \end{array}$$

2.2 Semantics

Since a formula will be considered as a part of a process, then the semantics of the logic will be given within the operational semantics of processes. In other word, in order to understand when a process P respects a formula Φ, the reader needs to understand how the process (P, Φ) evolves. Intuitively, a process respects a formula (the process (P, Φ) can evolve) if the first action of the process respects the first proposition of the formula and the residual part of the process respects the residual part of the formula.

2.3 Examples

- $\Phi = send(x) : x > 3$: A process respects this formula if it starts by sending a value greater than 3.
- $\Phi = (\neg read(x))^*((\neg send(y))^* ff)$: A process respects this formula if it does not perform a send on the network after reading some data.

3 Program Specification

Hereafter, we give the syntax and the semantics of the language that we use to specify programs. Basically, it is a modified version of the the "Basic Process Algebra" (BPA) [12]. This new algebra will be denoted in the rest of this paper by BPA^Φ.

3.1 Syntax

As shown by Table 2, the syntax of BPA^Φ is similar to the one of BPA except that we introduce new form of process which is (P, Φ). Basically, a process is a combination of some operators, atomic actions and other processes according to some rules. The constants 0 and δ represent respectively a successful termination and anormal termination. Atomic actions will be designed by letters $a, b, etc.$. The operator "." represents the sequential composition. A process $P_1.P_2$ has to run P_1 until it terminates, and then behaves as P_2. Also, the operator "+" represents the alternative composition. A process $P_1 + P_2$ has a choice to execute the process P_1 or the process P_2. To represent iteration and infinite processes we use the Kleene star operator denoted by "*". A process $P_1^*P_2$ behaves as P_2 if P_1 is not able to evolve otherwise it behaves as $P_1.P_1^*P_2$. Finally, the process (P, ϕ), where P is a process and Φ is a formula in L_φ, can be seen as the process P controlled by Φ. Note that for this paper

we suppose that the different APIs functions are actions such as : $read(x)$, $write(x)$, $readFile(y)$, etc. and assigning a value of an expression to some variable, $x = e$. We suppose also that all variables of processes ranges over a set $\mathcal{X}$

Table 2: Syntax of BPA^{ϕ}.

$P ::= 0 \mid \delta \mid a \mid P_1.P_2 \mid P_1 + P_2 \mid P_1^* P_2 \mid (P, \phi)$

3.2 Semantics

Hereafter, we define the semantics of our language in an operational way. As shown by Table 5 the definition of the transition relation $\rightarrow$ of BPA^{Φ} is based on different kind of relations such as $_{\downarrow}$, $\equiv$ and $\models$ that are defined hereafter.

Normal Form ($_{\downarrow}$) As we have stated before, a process (P, Φ) can evolve only if the first action of P respects the first proposition of Φ and the residual part of P respects the residual part of Φ. Therefore, we need somehow to know what are the first possible actions of P and the first proposition of Φ.

3.2.1 Definition.[Normal form]
A formula ϕ (resp. a process P) is called in normal form iff it has the following form :

$$\phi_{\downarrow} = \bigvee_{1 \leq i \leq n} p_i.\varphi_i \qquad \text{(resp. } P_{\downarrow} = \sum_{1 \leq i \leq n} a_i.P_i + \sum_{1 \leq j \leq m} b_j\text{)}$$

where p_i is a proposition and φ_i is a formula

Simplification relation ($\equiv$) In order to simplify the operational semantics of BPA^{Φ}, we introduce the relation $\equiv$ defined by Table 8. Most of the axioms $A_1 \ldots A_5$) of the Table 8 are presented in [12]. The axiom A_6 expresses the fact that a process $((P, \phi_1), \phi_2)$ can evolve only if P satisfies both ϕ_1 and ϕ_2.

Table 3: Axiomes of BPA^{ϕ}.

$P_1 + P_2$	$\equiv$	$P_2 + P_1$	(A_1)	$P + 0$	$\equiv$	P	(A_2)
$P + \delta$	$\equiv$	P	(A_3)	$\delta.P$	$\equiv$	δ	(A_4)
$0.P$	$\equiv$	P	(A_5)	$((P, \phi_1), \phi_2)$	$\equiv$	$(P, \phi_1 \wedge \phi_2)$	(A_6)

Environment (ξ) The semantics of a process (P, ϕ) is defined according an environment $\xi = (\Gamma, \Delta)$, where:

- Γ is a store (a memory) that links all variables of the evaluated process to values. It should be seen as a mapping that attributes a value to a variable (i.e., $\Gamma : \mathcal{X} \longrightarrow \mathcal{Z}$). If a is an atomic action in the process P, then $a\Gamma$ denotes the same action where variables are substituted by their values in Γ.

- Δ is also a formula environment (a mapping). To each variable in a given formula we attribute the set of values that could be attributed to this variable (i.e., $\Delta : \mathcal{X} \longrightarrow \wp(\mathcal{Z})$).
  - If for $x \in \mathcal{X}$ we have $\Delta(x) = \mathcal{Z}$, then Δ will be denoted by $\top$.
  - If for $x \in \mathcal{X}$ we have $\Delta(x) = \emptyset$, then Δ will be denoted by $\bot$.
  - If there exists $x \in \mathcal{X}$ such that $\Delta(x) = \emptyset$, then we say that $\Delta \approx \bot$.
  - A formula environment Δ^1 is generally reduced to its significant part which are the variables that are not attached to $\mathcal{Z}$.
  - If Δ^1 et Δ^2 are two formulas environment, then $\Delta^1 \cup \Delta^2 = \bigcup_{x \in \mathcal{X}} \{x \mapsto \Delta^1(x) \cup \Delta^2(x)\}$.
  - If Δ^1 et Δ^2 are two formulas environment, then $\Delta^1 \cap \Delta^2 = \bigcup_{x \in \mathcal{X}} \{x \mapsto \Delta^1(x) \cap \Delta^2(x)\}$.

Evaluation functions To simplify the presentation of the operational semantics of BPA^{ϕ}, we suppose that the semantics of boolean expressions and elementary program's actions are given by the following tow functions :

- $[\![-]\!]^{\Gamma}_{\mathcal{B}} : \mathcal{B} \rightarrow (\Gamma \rightarrow \{\mathrm{tt}, \mathrm{ff}\})$: this function gives semantics of a boolean condition under an environment Γ. It takes a boolean condition and returns a function that takes an environment and returns a boolean value.
- $[\![-]\!]^{\Gamma}_{\mathcal{A}} : \mathcal{A} \rightarrow (\Gamma \rightarrow \Gamma)$: this function takes an action and returns a function that takes environment and returns a new environment.

Satisfaction Relation ($\models$) Since a process (P, Φ) can evolve only if the first action of P respects the first proposition of Φ, then we need to define when an atomic action satisfies a proposition (or a formula in general). For that reason, we introduce the satisfaction relation $\models$. The relation $\models$ takes an action a, a store Γ, and a formula ϕ and returns an environment formula Δ showing under which restrictions $a\Gamma$ respects the formula ϕ. The definition of "$\models$" is given by Table 4.

Here are some examples:

- $send(x), [x \mapsto 3] \models send(y), [y \mapsto \{3\}]$.
- $send(3), \Gamma \models \neg read(x), \top$.
- $send(3), \Gamma \models \neg send(x), \bot$.

Transition Relation The operational semantics of BPA^{ϕ} (see Table 5) is defined by the relation $\longrightarrow \subseteq \mathcal{C} \times \mathcal{A} \times \mathcal{C}$, where $\mathcal{C}$ is the set of configurations ($\mathcal{C} = \mathcal{P} \times \Upsilon$, where $\mathcal{P}$ is the set of processes and Υ is the set of environments) and $\mathcal{A}$ is the set of actions.

4 Formal monitor

The main aim of this work is to ensure that a program P will not violate a given security policy Φ at run time. To achieve this goal, we have just to execute the process (P, Φ). In fact, the semantics of our language was defined in such a way that the program (P, Φ) can evolve only when the security policy is respected otherwise the program is interrupted. Note also that our language allows us to easily attach a formula to any slice of a given program. In fact, if, for instance, $P_1.P_2$ is a process then $(P_1, \Phi).P_2$ is also a process where the slice P_1 has to respects the formula Φ. Different formulas can also be attached to different slices.

Since some parts of the formula could generally be verified statically, therefore it will be interesting to "eliminate" from a formula all what can be verified statically. For instance if $P = send(2).read(y).send(y)$ and Φ is $send(x).\neg read(5).send(y)$, then (P, Φ) can be simplified to the process $send(2).(read(y), \neg read(5)).send(y)$. The optimisation issue is addressed by the following section.

Table 4: The satisfaction relation $\vDash$.

$\frac{[\![b]\!]^{\Gamma}_{\mathcal{B}} = tt}{a, \Gamma \vDash b, \top}$	$\frac{[\![b]\!]^{\Gamma}_{\mathcal{B}} = ff}{a, \Gamma \vDash b, \bot}$
$\frac{\exists \sigma \mid a\Gamma = f\sigma \wedge [\![b\sigma]\!]^{\Gamma}_{\mathcal{B}} = tt}{a, \Gamma \vDash f : b, \cup_{[x \mapsto v] \in \sigma} \{x \mapsto \{v\}\}}$	$\frac{(\forall \sigma \mid a\Gamma \neq f\sigma) \vee (\exists \sigma \mid a\Gamma = f\sigma \wedge [\![b\sigma]\!]^{\Gamma}_{\mathcal{B}} = ff)}{a, \Gamma \vDash f : b, \bot}$
$\frac{\exists \sigma \mid a\Gamma = f\sigma \wedge [\![b\sigma]\!]^{\Gamma}_{\mathcal{B}} = tt}{a, \Gamma \vDash \neg f : b, \bot}$	$\frac{(\forall \sigma \mid a\Gamma \neq f\sigma) \vee (\exists \sigma \mid a\Gamma = f\sigma \wedge [\![b\sigma]\!]^{\Gamma}_{\mathcal{B}} = ff)}{a, \Gamma \vDash \neg f : b, \top}$
$\frac{a, \Gamma \vDash \phi_1, \Delta^1 \quad a, \Gamma \vDash \phi_2, \Delta^2}{a, \Gamma \vDash \phi_1 \vee \phi_2, \Delta^1 \cup \Delta^2}$	$\frac{a, \Gamma \vDash \neg\phi_1 \wedge \neg\phi_2, \Delta}{a, \Gamma \vDash \neg(\phi_1 \vee \phi_2), \Delta}$
$\frac{a, \Gamma \vDash \phi_1, \Delta^1 \quad a, \Gamma \vDash \phi_2, \Delta^2}{a, \Gamma \vDash \phi_1 \wedge \phi_2, \Delta^1 \cap \Delta^2}$	$\frac{a, \Gamma \vDash \neg\phi_1 \vee \neg\phi_2, \Delta}{a, \Gamma \vDash \neg(\phi_1 \wedge \phi_2), \Delta}$
$\frac{a, \Gamma \vDash p, \Delta}{a, \Gamma \vDash p.\phi, \Delta}$	$\frac{a, \Gamma \vDash \phi, \Delta}{a, \Gamma \vDash \neg\neg\phi, \overline{\Delta}}$
$\frac{a, \Gamma \vDash (\phi_1.\phi_2)_{\downarrow}, \Delta}{a, \Gamma \vDash \phi_1.\phi_2, \Delta}$	$\frac{a, \Gamma \vDash (\neg(\phi_1.\phi_2))_{\downarrow}, \Delta}{a, \Gamma \vDash \neg(\phi_1.\phi_2), \Delta}$
$\frac{a, \Gamma \vDash (\phi_1^*\phi_2)_{\downarrow}, \Delta}{a, \Gamma \vDash \phi_1^*\phi_2, \Delta}$	$\frac{a, \Gamma \vDash (\neg(\phi_1^*\phi_2))_{\downarrow}, \Delta}{a, \Gamma \vDash \neg(\phi_1^*\phi_2), \Delta}$

5 Security Enforcement Simplification by Typing

In this section, we define a typing system that aims to statically optimize a program by removing all useless tests (tests that can be statically evaluated). A jugement of the typing system has the following form:

$$\mathcal{H} \vdash P : \tau, \Delta$$

where

- $\mathcal{H}$ is an environment used to handle recursive programs so that their typing will always terminate. The initial value of this environment is generally the empty set.
- P is the initial program that we want to optimize.
- τ is the type of P which is it's optimized version.
- Δ is a formula environment used to propagate values of instantiated variables in formulas.

Note that, for the sake of simplicity, the typing system handles only a subset of BPA^{ϕ} that has the same expressivity of the the complete BPA^{ϕ}. The difference is related only to recursive processes and recursive formulas. With the subset of processes handle by the typing system, we consider only recursive processes that have the form $P^*\delta$ and recursive formulas that have the form $\Phi^* ff$, where P and Φ do not contain the operator $*$. It has been

Table 5: Operational semantics of BPA^{ϕ}.

$$(R_{\equiv})\ \frac{P \equiv P_1 \quad P_1, \xi \xrightarrow{a} P_2, \xi' \quad P_2 \equiv Q}{P, \xi \xrightarrow{a} Q, \xi'}$$

$$(R^{a})\frac{\Box}{a, (\Gamma, \Delta) \xrightarrow{a} 0, ([\![a]\!]^{\Gamma}, \Delta)} \qquad (R^{a}_{\phi})\frac{a, \Gamma \vDash \phi, \Delta'}{(a, \phi), (\Gamma, \Delta) \xrightarrow{a} 0, (\Gamma, \Delta' \cap \Delta)}\ \Delta' \cap \Delta \not\approx \bot$$

$$(R^{a}_{\neg\phi})\frac{a, \Gamma \vDash \phi, \Delta'}{(a, \phi), (\Gamma, \Delta) \xrightarrow{\delta} \delta, (\Gamma, \bot)}\ \Delta' \cap \Delta \approx \bot \qquad (R^{P}_{\phi})\frac{(P, \phi)_{\downarrow}, \xi \xrightarrow{a} P_2, \xi'}{(P, \phi) \xrightarrow{a} P_2, \xi'}(P \neq a)$$

$$(R_{+})\frac{P_1, \xi \xrightarrow{a} P', \xi'}{P_1 + P_2, \xi \xrightarrow{a} P', \xi'}\ a \neq \delta \qquad (R_{.})\frac{P_1, \xi \xrightarrow{a} P', \xi'}{P_1.P_2, \xi \xrightarrow{a} P'.P_2, \xi'}$$

$$(R_{*})\frac{P_1, \xi \xrightarrow{a} P', \xi'}{P_1^{*}P_2, \xi \xrightarrow{a} P'.(P_1^{*}P_2), \xi'} \qquad (R^{d}_{*})\frac{\nexists\, a : P_1, \xi \xrightarrow{a} P'', \xi'' \quad P_2, \xi \xrightarrow{a} P', \xi'}{P_1^{*}P_2, \xi \xrightarrow{a} P', \xi'}$$

shown, in [7, 8], that the form $P^{*}\delta$, where P is $*$ free, is as expressive as $P_1^{*}P_2$, meaning that every program can be transformed to an equivalent one that contains at most one loop which is not followed by any instruction.

The Typing rules are given by Table 6 where the function TypeOf(a, ϕ) is defined as following:

$$\text{TypeOf}(a, \phi) \quad = \quad \begin{cases} (\delta, C), \bot & \text{If} \quad a \Vdash \phi, (C, \bot) \\ (a, C), \xi & \text{If} \quad a \Vdash \phi, (C, \xi) \text{ and } \xi \not\equiv \bot \end{cases} \tag{1}$$

where $\Vdash$ is defined by Table 7. Note that $C(\sigma)$ and $\neg C(\sigma)$ are two boolean conditions extracted for σ as following:

$$\begin{array}{lcl} C(\varepsilon) & = & tt \\ C(\{x \mapsto t\} \cup \sigma) & = & (x = t) \wedge C(\sigma) \end{array} \quad \Big\| \quad \begin{array}{lcl} \neg C(\varepsilon) & = & ff \\ \neg C(\{x \mapsto t\} \cup \sigma) & = & (x! = t) \vee \neg C(\sigma) \end{array}$$

Note that the program returned by the typing system can be considerable simplified using the rewriting rules of table 8.

6 Example

Before presenting the example, it will be interesting to make a link between our language and the usual notation used within imperative languages. This link is summarized by Table 9. Note also that $(send(x), \neg send(3))$ is equivalent representation of $(send(x), x! = 3)$ since $TypeO(send(x), \neg send(3)) = (send(x), x! = 3)$.

6.1 Exemple 1

- Program written in usual imperative language notation:

Table 6: Typing System.

$$(S_0)\frac{\square}{\mathcal{H} \vdash 0 : 0, \top} \qquad (S_{0\phi})\frac{\square}{\mathcal{H} \vdash (0,\phi) : 0, \top}$$

$$(S_\delta)\frac{\square}{\mathcal{H} \vdash \delta : \delta, \bot} \qquad (S_{\delta\phi})\frac{\square}{\mathcal{H} \vdash (\delta,\phi) : \delta, \bot}$$

$$(S_{act})\frac{\square}{\mathcal{H} \vdash a : a, \top} \qquad (S_{act\phi})\frac{\square}{\mathcal{H} \vdash (a,\phi) : \text{TypeOf}(a,\phi)}$$

$$(S_*)\frac{\mathcal{H} \vdash P : \tau, \Delta}{\mathcal{H} \vdash P^*\delta : \tau^*\delta, \Delta} \qquad (S_\phi)\frac{\mathcal{H} \vdash (P,\phi)_\downarrow : \tau, \Delta}{\mathcal{H} \vdash (P,\phi) : \tau, \Delta} P \neq a$$

$$(S_.)\frac{\mathcal{H} \vdash P_1 : \tau_1, \Delta_1 \quad \mathcal{H} \vdash P_2 : \tau_2, \Delta^2}{\mathcal{H} \vdash P_1.P_2 : \tau_1.\tau_2, \Delta^1 \cap \Delta^2} \Delta^1 \cap \Delta^2 \not\approx \bot \qquad (S_\bot)\frac{\mathcal{H} \vdash P_1 : \tau_1, \Delta^1 \quad \mathcal{H} \vdash P_2 : \tau_2, \Delta^2}{\mathcal{H} \vdash P_1.P_2 : \delta, \bot} \Delta^1 \cap \Delta^2 \approx \bot$$

$$(S_+)\frac{\mathcal{H} \vdash P_1 : \tau_1, \Delta^1 \quad \mathcal{H} \vdash P_2 : \tau_2, \Delta^2}{\mathcal{H} \vdash P_1 + P_2 : \tau_1 + \tau_2, \Delta^1 \cup \Delta^2} \qquad (S_{\mathcal{H}})\frac{\square}{\mathcal{H} \cup \{P \mapsto \tau\} \vdash P : \tau, \varepsilon}$$

$$(S_{**})\frac{\mathcal{H}\dagger[(P,0)] \vdash (P_1.P_1^*\delta, \phi.\phi^* ff) : \tau, \Delta}{\mathcal{H} \vdash \underbrace{(P_1^*\delta, \phi^* ff)}_{P} : \tau^*\delta, \Delta} \qquad (S_{*\phi})\frac{\mathcal{H} \vdash (P_1^*\delta, \phi^* ff) : \tau, \Delta}{\mathcal{H} \vdash \underbrace{(P_1^*\delta, \phi^* ff.\phi')}_{P} : \tau, \Delta} \forall \tau' : (P,\tau') \notin \mathcal{H}$$

```
while(true)
do
  read(x);
  if(x = Pwd )
      read(y);
      send(y);
  else
      send(x);
endDo
```

- Equivalent program in BPA^Φ:

$$\underbrace{(r(x).((r(y).s(y), x = Pwd) + (s(x), x \neq Pwd)))}_{P}{}^*\delta$$

  where r denotes the $read$, s denotes the $send$ and p denotes the $print$ action.

- Formula ϕ : $(read(x).\neg send(3))^* ff$ meaning that a read action can never be followed by $send(3)$

- The result of the typing (the proof is omitted due to the lack of space) of (P, Φ) is:

Table 7: The satisfaction relation $\Vdash$.

$$\frac{\exists\sigma \mid \sigma = mgu(a,f)}{a \Vdash f : b, (C(mgu(a,f\sigma)), \bigcup_{[x\mapsto t]\in mgu(a\sigma,f)} \{x \mapsto \{t\}\})}$$

$$\frac{\exists\sigma \mid \sigma = mgu(a,f)}{a \Vdash \neg f : b, (\neg C(mgu(a,f\sigma)), \top)} \qquad \frac{}{a \Vdash b, (\varepsilon, \top)}$$

$$\frac{\forall\sigma : a\sigma \neq f\sigma}{a \Vdash \neg f : b, \varepsilon, \top} \qquad \frac{\forall\sigma : a\sigma \neq f\sigma}{a \Vdash f : b, (\varepsilon, \bot)}$$

$$\frac{a \Vdash \phi_1, (C_1, \Delta_1) \quad a \Vdash \phi_2, (C_2, \Delta_2)}{a \Vdash \phi_1 \vee \phi_2, (C_1 \vee C_2, \Delta_1 \cup \Delta_2)} \qquad \frac{a \Vdash \neg\phi_1 \wedge \neg\phi_2, \Gamma}{a \Vdash \neg(\phi_1 \vee \neg\phi_2), \Gamma}$$

$$\frac{a \Vdash \phi_1, (C_1, \Delta_1) \quad a \Vdash \phi_2, (C_1, \Delta_1)}{a, \Vdash \phi_1 \wedge \phi_2, (C_1 \wedge C_2, \Delta_1 \cap \Delta_2)} \qquad \frac{a \Vdash \neg\phi_1 \vee \neg\phi_2, \Gamma}{a \Vdash \neg(\phi_1 \wedge \phi_2), \Gamma}$$

$$\frac{a \Vdash p, \Gamma}{a \Vdash p.\phi, \Gamma} \qquad \frac{a \Vdash \phi, \Gamma}{a \Vdash \neg\neg\phi, \Gamma}$$

$$\frac{a \Vdash (\phi_1.\phi_2)_\downarrow, \Gamma}{a \Vdash \phi_1.\phi_2, \Gamma} \qquad \frac{a \Vdash (\neg(\phi_1.\phi_2))_\downarrow, \Gamma}{a \Vdash \neg(\phi_1.\phi_2), \Gamma}$$

$$\frac{a \Vdash (\phi_1^*\phi_2)_\downarrow, \Gamma}{a \Vdash \phi_1^*\phi_2, \Gamma} \qquad \frac{a \Vdash (\neg(\phi_1^*\phi_2))_\downarrow, \Gamma}{a \Vdash \neg(\phi_1^*\phi_2), \Gamma}$$

Table 8: Rewriting Rules

(a, tt)	$\rightarrow$	a	(a, ff)	$\rightarrow$	δ
$P + P$	$\rightarrow$	P	$P + 0$	$\rightarrow$	P
$P + \delta$	$\rightarrow$	P	$\delta.P$	$\rightarrow$	δ
$0.P$	$\rightarrow$	P	$P.0$	$\rightarrow$	P
$(a_1, \phi_1) + (a_1, \phi_2)$	$\rightarrow$	$(a_1, \phi_1 \vee \phi_2)$	$P.P^*\delta$	$\rightarrow$	$P^*\delta$
$a.P_1 + a.P_2$	$\rightarrow$	$a.(P_1 + P_2)$	$a_1.P + a_2.P$	$\rightarrow$	$(a_1 + a_2).P$

$$(r(x).((r(y).(s(y), y \neq 3), x = Pwd) + (s(x), x \neq Pwd \wedge x \neq 3)))^*\delta$$

- Optimized program written in usual imperative language notation :

```
while(true)
do
  read(x);
```

Table 9: Imperative Language vs BPA^*

Imperative Language	Processes
$P_1; P_2$	$P_1.P_2$
while c **do** P	$(P, c)^*\delta$
if c **then** P **else** Q	$(P, c) + (Q, \neg c)$

```
    if(x = Pwd )
        read(y);
        if(Y != 3)
            send(y);
    else
        if( x != 3)
            send(x);

endDo
```

6.2 Exemple 2

- We consider the same program as the one of the first example, but with the following formula.
- Formula $\Phi : (tt.\neg send(3)^* ff$ (tt means every action). Note that the simplicity of the formula is due to the lack of space and we want to show how we deal with loops.
- The result of the typing (the proof is omitted due to the lack of space) of (P, Φ) is:

$$(r(x).((r(y).s(y), x = Pwd)+(s(x), x \neq Pwd \wedge x \neq 3)).r(x).((r(y).(s(y), y \neq 3), x = Pwd)+(s(x), x \neq Pwd \wedge x \neq 3)))^*\delta$$

  Intuitively, we have a process of the form $(P^*\delta, \phi^* ff)$, then we can apply only the rule (S_{**}). This will add the process $(P^*\delta, \phi^* ff)$ to the static environment $\mathcal{H}$ that will ensure the termination of the typing procedure. In fact, after two iterations, as shown in the code below, we return to the initial process.

- Optimized program written in usual imperative language notation:

```
while(true)
do
  read(x);
  if(x = Pwd )
      read(y);
      send(y);
  else
```

```
        if( x != 3)
            send(x);
    read(x);
    if(x = Pwd )
        read(y);
        if(Y != 3)
            send(y);
    else
        if( x != 3)
            send(x);

endDo
```

7 Equivalence Theorem

The aim of this section is to prove that the optimized version (type) of any process is equivalent to its original version. The equivalence relation that we looking for, denoted by $\sim$, is the bisimulation. However, we prove our result for a more general relation, denoted by $\sim_{\mathcal{H}}$ (bisimulation modulo a well formed environment), which gives the requested result when $\mathcal{H}$ is an empty set. First let's define a well formed environment.

7.0.1 Definition.[Well Formed Environment]
Environment $\mathcal{H}$ is said to be well formed iff for all $(P, \tau) \in \mathcal{H}$, these two condition are satisfied :

1. P is of the form $(P_1^* \delta, \phi^* f f)$;
2. τ is equal to 0.

Now let's define $\sim_{\mathcal{H}}$.

7.0.2 Definition.[$\sim_{\mathcal{H}}$]
Let $\mathcal{H}$ a well formed environment. We define $\sim_{\mathcal{H}}$ as the biggest relation satisfying the following conditions: $P \sim_{\mathcal{H}} Q$ if :

1. (P,Q) $\in \mathcal{H}$, or
2. (i) If $P, \xi \xrightarrow{a} P', \xi'$ then $Q, \xi \xrightarrow{a} Q', \xi'$ and $P' \sim_{\mathcal{H}} Q'$, and
   (ii) If $Q, \xi \xrightarrow{a} Q', \xi'$ then $P, \xi \xrightarrow{a} P', \xi'$ and $Q' \sim_{\mathcal{H}} P'$

Note that when $\mathcal{H}$ is an empty set $\sim_{\mathcal{H}}$ is the bissimulation.

7.0.3 Theorem.
$\forall P \in BPA^{\Phi}, \forall \mathcal{H}$, a well formed environment, we have: $\mathcal{H} \vdash P : \tau, \Delta \Longrightarrow P \sim_{\mathcal{H}} \tau$

The proof of the previous theorem is omitted due to space restriction.

8 Conclusion and Future works

In this paper, we have defined a new process algebra that contains a special form of precess (P, ϕ). The semantic of the proposed algebra ensure that the program P can evolve only if it does not violate the security policy ϕ. Furthermore, we have defined a typing system that allows us to optimize a program by efficiently removing the tests required by the security and that can be verified statically. Finally, we have proved the correctness of optimization performed by the typing system tanks to the Theorem 7.0.3.

As a future work, we want to extend our logic L_ϕ to give to the end-user more flexibility to handle the case where the program reach a point at which it violates the security policy. In this paper, we decided to interrupt a program as soon as its security property is violated. However it is not necessarily the suitable decision for all application and it is better to give the chose to the end-user that specify the series of actions that he/she want to execute when the security property is violated (It's similar to the principle of exceptions in Java for example). To this end, we can extend L_φ by simply adding the form (ϕ, P). Intuitively, using this new formula, the program $(P_1, (\phi, P_2)$ behaves as P_1 until the formula ϕ is violated and then behaves like P_2.

References

[1] D. Grossman and J. G. Morrisett, Scalable Certification for Typed Assembly Language. *In Proc. of TIC '00: Selected papers from the Third International Workshop on Types in Compilation*, pages 117-146, 2001. Springer-Verlag.

[2] P. Cousot, An indo-french school on abstract interpretation 1em plus 0.5em minus 0.4emOverview, 1996.

[3] E. M. Clarke, O. Grumberg and D. A. Peled, Model Checking. MIT Press, 2000.

[4] L. Bauer,J. Ligatti and D. Walker, More Enforceable Security Policies. *In Proc. of the FLoC'02 workshop on Foundations of Computer Security*, pages 95-104, 2002. Iliano Cervesato.

[5] Fred B. Schneider, Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, volume 3, pages 30-50, 2000. ACM Press.

[6] E.A. Emerson, *Temporal and modal logic*. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B*, pages 995-1072, 1990.

[7] D. Kozen, Kleene Algebra with Tests. *ACM Transactions on Programming Languages and Systems, volume 19*, pages 427-443, 1997.

[8] M. Mirkowska, Algorithmic logic and its applications. PhD thesis, Warsaw school, 1972.

[9] A. R. Twyman, Flexible code safety for Win32. Msc thesis, Massachusetts Institute of Technology, 1999.

[10] Fred B. Schneider and U. Erlingsson, SASI Enforcement of Security Policies: A Retrospective. *In Proc. of New Security Paradigms Workshop*, 1999. ACM Press.

[11] D. Walker, A type system for expressive security policies. *In Proc. of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, , pages 254-267, 2000. ACM Press.

[12] J. C. M. Baeten and W. P. Weijand, Process Algebra. Cambridge University Press, 1990.

Session III

Information Flow

Unifying Confidentiality and Integrity in Downgrading Policies

Peng Li Steve Zdancewic
Department of Computer and Information Science
University of Pennsylvania
{lipeng,stevez}@cis.upenn.edu

Abstract

Confidentiality and integrity are often treated as dual properties in formal models of information-flow control, access control and many other areas in computer security. However, in contrast to confidentiality policies, integrity policies are less formally studied in the information-flow control literature. One important reason is that traditional noninterference-based information-flow control approaches give very weak integrity guarantees for untrusted code. Integrity and confidentiality policies are also different with respect to implicit information channels.

This paper studies integrity downgrading policies in information-flow control and compares them with their confidentiality counterparts. We examine the drawbacks of integrity policies based on noninterference formalizations and study the integrity policies in the framework of downgrading policies and program equivalences. We give semantic interpretations for traditional security levels for integrity, namely, tainted and untainted, and explain the interesting relations between confidentiality and integrity in this framework.

Keywords: language-based security, information flow, integrity, downgrading, security policy.

1 Introduction

Language-based information-flow security [11] provides end-to-end guarantees on the dependency and the propagation of information in the system, which is usually formalized as *noninterference* [2, 5] properties. Such security guarantees are ideal for protecting *confidentiality*, where secret information is not permitted to propagate to public places. On the other hand, information-flow control can also be used to provide *integrity* guarantees, where important data in the system is not allowed to be affected by untrusted sources of information. Confidentiality and integrity can be viewed as duals [1] in many areas of computer security. In information flow, confidentiality policies prevent secret data from being leaked out to the adversary, while integrity policies restrict the use of data coming from the adversary.

There are very practical applications of information-flow policies for *integrity*. For example, an unsafe script on the web server could use strings from untrusted inputs to compose a SQL query string and then have the database management system execute the query, which potentially allow the attacker to execute commands in the database. The Perl programming language provides built-in support for dynamic information-flow checking. Data from user inputs and the network is marked as tainted, while system calls require untainted data. Tainted data can be converted to untainted data through pattern matching, which effectively forces the programmer to examine untrusted data and avoid malicious attacks. Code analysis tools such as cqual [12] perform static information-flow checking to detect the use of dangerous data. Such tools have been used to find bugs in large-scale software systems.

Despite the practical interests, integrity policies are often less formally studied in the literature of language-based information-flow security. Many formal studies are focused on confidentiality and merely mention that confidentiality and integrity are duals. In fact, confidentiality and integrity are not symmetric in traditional approaches based on noninterference. The noninterference property is too emphasized for confidentiality and it is not appropriate for integrity. For example, noninterference does not give useful integrity guarantees for untrusted code, and it is often too strong for practical use because it rules out all implicit information flows, most of which are not harmful. Noninterference also does not handle downgrading. Section 2 identifies these challenges on integrity policies in information-flow.

To fix the aforementioned weaknesses of noninterference-based integrity policies, we need alternative formalizations for information flow. Out recent research [4] uses downgrading policies and program equivalences to formalize the goals of language-based information-flow security. The original motivation is to achieve an end-to-end security guarantee like noninterference when *downgrading* (or declassification) is available in the system. While this framework was originally focused on confidentiality, it also provides a basis for very expressive integrity downgrading policies. Moreover, it is possible to achieve a better formal security goal for integrity that avoids the drawbacks of noninterference-based definitions. Section 3 extends the framework of downgrading policies with integrity policies. We discuss how to formalize the security goal for integrity present a highly symmetric view over confidentiality and integrity in this framework.

2 Challenges of Integrity

2.1 Policy expressiveness

The formal definition of noninterference gives an intuitive and absolute meaning of confidentiality, but its relationship with integrity is less straightforward. In general, integrity has many meanings in computer security. For example, Pfleeger's security textbook [8] describes integrity policies that require that: data is modified only by authorized principals, data is modified in permitted ways, data is consistent, valid, meaningful and correct, etc. The actual meaning of integrity depends on the specific context. Noninterference only provides a particular kind of integrity guarantee, that is, trusted data is not affected by the propagation of untrusted data. Apparently, there are many information integrity policies that noninterference cannot express. Most useful integrity policies involve accurate description of the actual computation. Integrity policies should not only specify who modified the data, but also specify how the data is manipulated.

2.2 Untrusted code

For untrusted code, noninterference gives a strong and practically useful guarantee for data confidentiality. This makes information-flow control a killer application for safely executing untrusted programs while giving them accesses to secret information.

However, traditional noninterference gives almost no integrity guarantee for untrusted code. The reason is that, when the code is not trusted, the adversary can manipulate trusted data in arbitrary ways in the program. For example, suppose the following function `foo` is written by an adversary. It takes two input arguments, performs some computation and returns a `untainted` value.

```
untainted int foo( untainted int a, tainted int b) { return a-a+0xff00; }
```

Although `foo` satisfies the noninterference policy, i.e. there is no information flow from the `tainted` input `b` to the `untainted` result, the result is not at all trustworthy because the adversary can return any arbitrary value in this function. Therefore, the data coming from untrusted programs (or software modules) must always be treated as tainted. Integrity policies based on noninterference definitions can only used in trusted environments,

where the programmers are cooperative and goal is to prevent accidental security exploits in trusted code. For the same reason, the two-dimensional *decentralized label model* [7] for confidentiality falls back to a one-dimensional model for integrity [3], which makes the integrity labels much less expressive in languages with information-flow type systems, such as FlowCaml [13, 9, 10] and Jif [6].

2.3 Downgrading

Pure noninterference is too ideal for practical applications. Most of the time, we do need to use information from untrusted sources in trusted places, as long as the tainted data can are verified to be safe. In the example of taint-checking mode in Perl, tainted data can be converted to untainted using pattern-matching. Clearly, there can be information propagation from untainted data to tainted, and noninterference policies are not directly applicable. This is the dual case for confidentiality, where secret data also needs to be declassified. This paper extends the framework of *downgrading policies* [4] and presents a symmetrical version of integrity downgrading policies, sometimes called *endorsement*.

2.4 Implicit information flow

Most confidentiality policies do not tolerate implicit information leakage. There are many implicit information channels such as control flow, timing channels and various side-effects that must be considered when untrusted code is available. For example, the following code has an implicit information leak from `secret` to `x` via control flow. If the code is not trusted and `x` is a publicly visible, the adversary can easily know the last bit of `secret` by observing the value of `x`. Noninterference policies rules out such implicit flows.

```
if (secret%2=1) then x:=1 else x:=0;
```

A straightforward solution is to use *downgrading* on the branching conditions where implicit flows are allowed. However, such implicit information propagation is almost always acceptable for data integrity policies, where the programmers are trusted and the goal is to prevent accidental destruction of trustworthy data. For example, the taint-checking mode in Perl does not check implicit information flows at all. Since the value of any trustworthy data can be directly modified by the programmer without violating information-flow policies as we have shown in the `foo` function above, there are few reasons to prevent implicit information flows, which are much more difficult to exploit to cause damage. Therefore, the security policy for protecting integrity does not have to be as strict as pure noninterference policies. Implicit information flow should be allowed by default, without the awkwardness of using explicit downgrading mechanisms.

3 Downgrading Policies for Integrity

To avoid the drawbacks of noninterference-based integrity policies, we study them in an alternative formal framework. Our recent research [4] uses downgrading policies and program equivalences to formalize the security goals of language-based information-flow security. This framework was originally focused on confidentiality, and this paper extends the integrity aspect of it. Similar to confidentiality labels, we define a partial ordering on integrity labels, formalizes the downgrading relation for integrity, and give interpretations to traditional security levels such as *untainted* and *tainted*. To highlight the symmetry between confidentiality and integrity, the definitions for two kinds of policies are given in parallel for the rest of the paper.

Briefly, this framework uses *downgrading policies* to express security levels of data and define the ordering among these security levels, which generalizes the simple security lattices $\mathsf{public} \sqsubseteq \mathsf{secret}$ for confidentiality and $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$ for integrity. A security level is simply a non-empty set of downgrading policies, where

each policy describes the computation related to downgrading. We reason about the programs in an end-to-end fashion. Each program takes input data and produces output data. Confidentiality policies are specified for each program input; integrity policies are specified for the program output. The security goal is then formalized using such security policies.

3.1 Downgrading policies and security labels

type	$\tau ::=$	$\mathsf{int} \mid \tau \to \tau$
constant	$c ::=$	$0, 1, 2, ...$
operator	$\oplus ::=$	$+, -, \%, =, ...$
downgrading policy	$m ::=$	$\lambda x\!:\!\tau.m \mid m\ m \mid x \mid c \mid\ \oplus\ \mid \mathsf{if}\ m\ m\ m$
confidentiality label	$cl ::=$	$\{m_1, \ldots, m_k\} \mid \mathsf{secret}_\tau \mid \mathsf{public}_\tau$
integrity label	$il ::=$	$\{m_1, \ldots, m_k\} \mid \mathsf{tainted}_\tau \mid \mathsf{untainted}_\tau$

Figure 1: Downgrading Policies and Security Labels

The syntax of downgrading policies and security labels is shown in Figure 1. Each downgrading policy is a term in the simply-typed λ-calculus, extended with operators and constants. The policy language is intended to be a fragment of the full language for which equivalence is decidable, so that the primitive operators will match those in the full language. Each downgrading policy represents some computation associated with downgrading as following:

- Confidentiality policies: each policy is a function that specifies how the data can be released to the public in the future. When this function is applied to the annotated data, the result is considered as public. For example, if a secret variable x is annotated with the confidentiality policy $\lambda x.\, x\%2$, it means the last bit of x can be released to public.
- Integrity policies: each policy is a term that specifies how the data has been computed in the past. For example, the integrity policy "2" means the data must be equal to 2 and works like a singleton type. The policy can be a function, too. For example, the policy $\lambda x.\, x\%10$ for an integer means that the integer must have been computed by $x\%10$, where x is potentially untrusted and we do not know what x is. Another useful policy is $\lambda x.\, \lambda y.\, \mathsf{match}(x, y)$, where match is a predefined pattern matching function and the policy means the data is the result of pattern matching. The weakest policy is the identity function, $\lambda x.\, x$, which simply gives no information about the how the data has been computed in the past.

Policies are typed in the simply-typed λ-calculus using the judgment $\Gamma \vdash m : \tau$. We use standard β-η equivalences $\Gamma \vdash m_1 \equiv m_2 : \tau$ for policy terms. Policy terms can be composed as functions using the following definitions.

Definition 3.1.1 (Policy composition) *If $\Gamma \vdash m_1 : \tau_1 \to \tau_3$ and $\Gamma \vdash m_2 : \tau_2 \to \tau_1$, the composition of m_1 and m_2 is defined as $m_1 \circ_\Gamma m_2 \stackrel{\triangle}{=} \lambda x\!:\!\tau_2.m_1\ (m_2\ x)$*

Definition 3.1.2 (Multi composition) *If $\Gamma \vdash m_1 : \tau \to \tau'$ and $\Gamma \vdash m_2 : \tau_1 \to \ldots \to \tau_k \to \tau$, the multi-composition of m_1 and m_2 is defined as $m_1 \odot_\Gamma m_2 \stackrel{\triangle}{=} \lambda x_1\!:\!\tau_1. \ldots \lambda x_k\!:\!\tau_k.m_1(m_2\ x_1 \ldots x_k)$*

Given the definition of downgrading policies, we can define a security label as a non-empty set of downgrading policies. We slightly abuse the notation to use cl to range over confidentiality labels and il to range over integrity labels. Labels are well-formed with respect to the type of data it annotates.

Definition 3.1.3 (Label wellformedness)
$\boxed{\vdash cl \triangleleft \tau} \iff \forall m \in cl, \exists \tau_1, \vdash m : (\tau \to \tau_1)$
$\boxed{\vdash il \triangleleft \tau} \iff \forall m \in il, \exists \tau_1, \ldots, \exists \tau_k, \vdash m : (\tau_1 \to \ldots \to \tau_k \to \tau)$

The meanings of two kinds of labels are symmetric:

- Confidentiality labels: each policy in the label can be used to declassify the data in the future. For example, if the value p is annotated with the confidentiality label $\{(\lambda p.\, \lambda x.\, p{=}x),\ (\lambda p.\, p\%2)\}$, it means that p can be declassified by comparing it with some other value, or by extracting its last bit.
- Integrity labels: each policy describes a possible computation that generated the value as the result in the past. For example, if the value p is annotated with the integrity label $\{(\lambda x.\, \mathsf{match}(x, c_1)), c_2\}$, it means the value is either the result of pattern-matching against pattern c_1 or a predefined constant c_2.

3.2 Label Ordering

Each label is syntactically represented as a set of downgrading policies, but the semantics of the label includes far more policies than explicitly specified. We define the interpretation of security labels as the following.

Definition 3.2.1 (Label interpretation)
$\mathbb{S}_\tau(cl) \stackrel{\triangle}{=} \{n' \mid n \in cl, \quad \vdash n' \equiv m \circ_\Gamma n : \tau_2\}$
$\mathbb{S}_\tau(il) \stackrel{\triangle}{=} \{n' \mid n \in il, \quad \vdash n : (\tau_1 \to \ldots \to \tau_k \to \tau),$
$\qquad\qquad \vdash n' \equiv (\lambda x_1 : \tau_1'. \ldots \lambda x_i : \tau_i'. n\ m_1 \ldots m_j) : (\tau_1' \to \ldots \to \tau_i' \to \tau)\ \}$

To understand the above definitions, suppose the security label has the policy n:

- For confidentiality: any function m composed with n is also a valid downgrading policy implied by n. The intuition is that if $(n\ x)$ is public, then $(m\ (n\ x))$ is also public, no matter what m is.
- For integrity: if we can compose n with some other terms and get a larger term n', in which n represent the final step of computation, then n' is also implied in this label, because any data computed by n' can also be treated as if it is computed by n using some input data.

  For example, suppose we have an integrity label $il_1 \stackrel{\triangle}{=} \{\lambda x.\, \mathsf{match}(x, c_1)\}$, then the following policies are also in $\mathbb{S}(il_1)$: $(\lambda a.\, \lambda b.\, \mathsf{match}(a + b, c_1))$, $(\mathsf{match}(c_2, c_1))$, etc. Intuitively speaking, if we only know that the data is the result of pattern-matching against the pattern c_1, then there are possibilities that the value matched with c_1 could be $a + b$, c_2, or any other values.

Based on the semantics of labels, we can easily define the ordering on security labels, using the set inclusion relation on label interpretations. We use the notation $l_1 \sqsubseteq l_2$ to say l_2 is a higher security level than l_1. The definitions for confidentiality and integrity labels are completely symmetric.

Definition 3.2.2 (Label ordering)
$\boxed{cl_1 \sqsubseteq cl_2 \triangleleft \tau} \iff \mathbb{S}_\tau(cl_1) \supseteq \mathbb{S}_\tau(cl_2), \quad \boxed{il_1 \sqsubseteq il_2 \triangleleft \tau} \iff \mathbb{S}_\tau(il_1) \subseteq \mathbb{S}_\tau(il_2).$

- Confidentiality: high security levels correspond to secret levels, low security levels corresponds to public levels. The intuition is that, each policy in the label corresponds to a path where secret data can be released to public places. The fewer paths there are, the more secure the data is.

- Integrity: high security levels correspond to tainted or untrusted levels and low security levels corresponds to trusted levels, because we would like to allow information flow from low levels to high levels but not in the other direction. The intuition is that, each policy corresponds to a possible computation that have generated the data. The more possibilities there are, the less trustworthy the data is.

We claim that the ordering of security labels generalizes the two-point lattices $\mathsf{public} \sqsubseteq \mathsf{secret}$ and $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$. In fact, all these traditional security levels can be interpreted in the framework of downgrading policies:

Definition 3.2.3 (Interpretation of special labels)
$\mathsf{secret}_\tau \triangleq \{\lambda x\!:\!\tau.\, 0\}$ $\quad \mathsf{public}_\tau \triangleq \{\lambda x\!:\!\tau.\, x\}$ $\quad \mathsf{tainted}_\tau \triangleq \{\lambda x\!:\!\tau.\, x\}$ $\quad \mathsf{untainted}_\tau \triangleq \{m \mid \vdash m\!:\!\tau\}$

- Confidentiality: we can prove that all the confidentiality labels of a given type form a lattice, where secret_τ is the top and public_τ is the bottom.

- Integrity: we can prove that $\mathsf{tainted}_\tau$ is the highest label of all the integrity labels. However, there is no single lowest label in the integrity ordering. Instead, there are many different lowest labels. For example, suppose $\tau = \mathsf{int}$, then $\{c_0\}$, $\{c_1\}$ and so on are all lowest labels. For a set of labels, their join (least upper bound) always exists, but they may not have a lower bound.

  The interpretation of $\mathsf{untainted}_\tau$ is the set of policies representing computations that do not use potentially untrusted inputs. We choose this interpretation for two reasons. First, it reflects the meaning of "untainted", i.e. the corresponding computation did not use tainted data. Second, its semantics is backward compatible with untainted in the traditional two-point security lattice, because when two untainted data meets together during computation (i.e. when computing $c_1 + c_2$), the result can also be labeled as untainted, which is the way things work in the two-point lattice. Thus, the ordering on integrity labels can be understood as a refined version of the two-point lattice $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$.

  Apparently, $\mathsf{untainted}_\tau$ does not provide a very strong security guarantee: data with this label can have any value. This coincides with the facts we mentioned in Section 2.2: if the code is not trusted, then untainted data can be anything. However, we now have security levels that provide more precise security guarantees: data with label $\{(\lambda x.\mathsf{match}(x, c_1))\}$ are guaranteed to match the pattern c_1; data with label $\{c_1, c_2\}$ is either c_1 or c_2, etc. Interestingly, the label $\{(\lambda x.\mathsf{match}(x, c_1))\}$ is not lower than untainted, but it provides a much more precise security guarantee than untainted does.

Overall, we can see that the label orderings for confidentiality and integrity are highly symmetric. The security levels public and tainted are both represented using the identity function and they both refer to data under control of the attacker. The security levels secret and untainted are also symmetric in some sense: secret are represented using constant functions, while untainted is represented using a set of terms that can be statically evaluated to normal forms.

The only asymmetry is that there are multiple lowest integrity labels, while there is only one highest confidentiality label. In fact, each lowest integrity label $\{c\}$ has its counterpart $\{\lambda x.\, c\}$ in the confidentiality lattice. It is just that all confidentiality labels $\{\lambda x.\, m\}$ such that x is free in m are structurally equivalent because their label interpretation are the same as the interpretation of a constant function. Intuitively speaking, different constant policies provide different integrity guarantees, but all constant policies have the same effect for confidentiality. This fact, together with the thoughts in Section 2.1, show a important difference between confidentiality and integrity in information flow. Confidentiality policies are destructive and do not care about the actual computation of secret data. If the secret data is destroyed and becomes garbage, it does not violate any confidentiality policies and the system is still secure. In contrast, integrity policies are highly related to the correctness and precision actual computation performed on the data.

3.3 Label Downgrading

The security label of data changes as the data is involved in some computation. We use the concept of *label downgrading* to describe the transition of security labels. To formalize this concept, suppose the data x_1 has type τ_1, and it is annotated by labels cl_1 and il_1. We use the concept of an *action* to model the computation on x_1: an action m on x_1 is a function applied to x_1. For example, suppose the computation on x_1 is $\mathsf{hash}(x_1)$, then the action is simply the hash function. If the computation is $x_1 + y$, then the action is $(\lambda x.\, x + y)$. Now, given the action m, suppose the result $(m\ x_1)$ has type τ_2, we can formally define the label cl_2 and il_2 on the result:

Definition 3.3.1 (Label Downgrading)

$$\boxed{(cl_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (cl_2 \lhd \tau_2)} \iff \vdash cl_1 \lhd \tau_1,\ \vdash cl_2 \lhd \tau_2,\ \forall m_2 \in cl_2, \exists m_1 \in \mathbb{S}_{\tau_1}(cl_1),\ \vdash m_2 \odot_\Gamma m \equiv m_1 : \tau$$

$$\boxed{(il_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (il_2 \lhd \tau_2)} \iff \vdash il_1 \lhd \tau_1,\ \vdash il_2 \lhd \tau_2,\ \forall m_2 \in il_2, \exists m_1 \in \mathbb{S}_{\tau_1}(il_1),\ \vdash m_2 \equiv m \odot_\Gamma m_1 : \tau$$

The judgment $(cl_1 \lhd \tau_1) \overset{m}{\rightsquigarrow} (cl_2 \lhd \tau_2)$ can be read as "the confidentiality label cl_1 at type τ_1 is transformed to a label cl_2 at type τ_2 under the action m". The definition is completely symmetric for confidentiality labels and integrity labels. Let us understand these rules by looking at some examples. For simplicity and readability, we omit the typing information in the following examples.

- Confidentiality labels: suppose we have the following labels and actions that are related using the label downgrading definition.

  $$cl_1 \triangleq \{\lambda x.\lambda y.\mathsf{hash}(x)\%4 = y\} \quad m_1 \triangleq \lambda x.\mathsf{hash}(x) \quad (cl_1 \lhd \mathsf{int}) \overset{m_1}{\rightsquigarrow} (cl_2 \lhd \mathsf{int})$$
  $$cl_2 \triangleq \{\lambda x.\lambda y.x\%4 = y\} \quad m_2 \triangleq \lambda x.x\%4 \quad (cl_2 \lhd \mathsf{int}) \overset{m_2}{\rightsquigarrow} (cl_3 \lhd \mathsf{int})$$
  $$cl_3 \triangleq \{\lambda x.\lambda y.x = y\} \quad m_3 \triangleq \lambda x.\lambda y.x = y \quad (cl_3 \lhd \mathsf{int}) \overset{m_3}{\rightsquigarrow} (\mathsf{public}_{\mathsf{int}} \lhd \mathsf{int})$$

  Suppose $x_2 \triangleq \mathsf{hash}(x_1)$, $x_3 \triangleq x_2\%4$ and $x_4 \triangleq (x_3 = p)$. If x_1 has label cl_1, then x_2 has label cl_2, x_3 has label cl_3, x_4 has label $\mathsf{public}_{\mathsf{int}}$. Intuitively speaking, the downgrading policies in a confidentiality label describe paths in which data can be downgraded in the future, which may involve several steps of computation. In the downgrading relation, the action m matches the prefix of such a path m_1, and the remaining path m_2 is preserved in the resulting label.

- Integrity labels: suppose we have the following labels, actions and downgrading relations:

  $$il_1 \triangleq \mathsf{tainted}_{\mathsf{int}} \quad m_1 \triangleq \lambda x.\mathsf{match}(x, c_1) \quad (il_1 \lhd \mathsf{int}) \overset{m_1}{\rightsquigarrow} (il_2 \lhd \mathsf{int})$$
  $$il_2 \triangleq \{\lambda x.\mathsf{match}(x, c_1)\} \quad m_2 \triangleq \lambda x.x + c_2 \quad (il_2 \lhd \mathsf{int}) \overset{m_2}{\rightsquigarrow} (il_3 \lhd \mathsf{int})$$
  $$il_3 \triangleq \{\lambda x.\mathsf{match}(x, c_1) + c_2\}$$

  Suppose $x_2 \triangleq \mathsf{match}(x_1, c_1)$ and $x_3 \triangleq x_2 + c_2$. If x_1 has label $\mathsf{tainted}_{\mathsf{int}}$, then x_2 has label il_2 and x_3 has label il_3. Intuitively speaking, the downgrading policies in an integrity label approximate the computation in the past, from which the data could have been computed. In the downgrading relation, the action m is appended to the history of computation m_1, and the result m_2 is in the resulting label.

3.4 Security Goals

The main question is: how to tell that a program is safe with respect to some security policies? We formalize the security goal in a language based on the simply-typed lambda calculus, which is basically an extension of our policy language. We define the security goals as end-to-end properties on the input-output relationship of the program.

Rather than using operational semantics, we use static program equivalences in the definition: if the program is safe, it must be equivalent to some special forms. The equivalence relation $\equiv$ is the standard $\beta - \eta$ equivalence, extended with some trivial rules for conditional expressions such as $e_1 \equiv \mathsf{if}\ 1\ e_1\ e_2$. The full definition of the equivalence relation is similar to those in our earlier work [4].

Definition 3.4.1 (Relaxed Noninterference) *Suppose the program uses secret input variables $\sigma_1, \sigma_2, \ldots$, where each input variable σ_i has a confidentiality label $\Sigma(\sigma_i)$ specified by the end user. For a program output e at type τ:*

- *e satisfies the confidentiality policy Σ, if $e \equiv f\ (m_1\ \sigma_{a_1}) \ldots (m_n\ \sigma_{a_n})$, where $\forall i.\sigma_i \notin FV(f)$ and $\forall j.m_j \in \Sigma(\sigma_{a_j})$.*
- *e satisfies the integrity policy il, if $e \equiv \begin{pmatrix} \mathsf{if}\ e_1(m_1\ e_{11}\ e_{12}\ ...) \\ \mathsf{if}\ e_2(m_2\ e_{21}\ e_{22}\ ...) \\ .... \\ \mathsf{if}\ e_n(m_n\ e_{n1}\ e_{n2}\ ...) \\ \qquad (m_0\ e_{01}\ e_{02}\ ...) \end{pmatrix}$ where $\forall j.m_j \in il$.*

The confidentiality condition requires that the program can be rewritten to a special form where secret variables are leaked to public places by using only the permitted functions (downgrading policies). Confidentiality policies are specified on the *input* of the program. The integrity condition requires that the program can be rewritten to a special form where the result is computed using one of the functions (downgrading policies) in the integrity label. Integrity policies are specified on the *output* of the program.

To understand the integrity guarantee better, consider the following two possibilities:

1. There is only one policy m in il. In this case, all the branches have the same m, so we have the following equivalence:

$$\begin{pmatrix} \mathsf{if}\ e_1(m\ e_{11}\ e_{12}\ ...) \\ \mathsf{if}\ e_2(m\ e_{21}\ e_{22}\ ...) \\ .... \\ \mathsf{if}\ e_n(m\ e_{n1}\ e_{n2}\ ...) \\ \qquad (m\ e_{01}\ e_{02}\ ...) \end{pmatrix} \equiv m \begin{pmatrix} \mathsf{if}\ e_1(e_{11}) \\ \mathsf{if}\ e_2(e_{21}) \\ .... \\ \mathsf{if}\ e_n(e_{n1}) \\ \qquad (e_{01}) \end{pmatrix} \begin{pmatrix} \mathsf{if}\ e_1(e_{12}) \\ \mathsf{if}\ e_2(e_{22}) \\ .... \\ \mathsf{if}\ e_n(e_{n2}) \\ \qquad (e_{02}) \end{pmatrix} ...$$

   This provides a very straightforward security guarantee. The attacker can only affect the result by *downgrading*, i.e. let untrusted data go through the downgrading policy m.

2. There are multiple policies in il. The body of each branch is still protected by a downgrading policy in il, but the attacker also has the ability to choose the exact branch to be taken, thus affecting the result via implicit control flow. Such implicit flows cannot be easily justified by *downgrading*, because the conditions $e_1 ... e_n$ are arbitrary programs not related to the downgrading policy. Our definition simply permits such implicit flows because we defined the integrity label as a set of *possible* computations. No matter which branch is taken, the final step of computation is always captured by the integrity label. This definition meets our requirement in Section 2.4. In contrast to the integrity condition, the confidentiality condition of Definition 3.4.1 does not tolerate any implicit information flow.

Definition 3.3.1 shows the symmetry between confidentiality and integrity conditions in a simple way: it describes how secret data are leaked to public places (for confidentiality) and how untrusted program generates trusted result (for integrity). However, it only specify policies on one side of the program and assumes that the program output is public and that the program inputs are tainted. Definition 3.3.1 can be further generalized to achieve more fine-grained end-to-end security conditions.

Definition 3.4.2 (Relaxed Noninterference (refined)) *Suppose the program uses input variables $\sigma_1, \sigma_2, \ldots$, where each input variable σ_i has a confidentiality label $\Sigma_c(\sigma_i)$ and integrity label $\Sigma_i(\sigma_i)$ specified by the end user. For a program output e at type τ:*

- *e satisfies the confidentiality policy cl, if $\forall n \in cl$, $(n\ e) \equiv f\ (m_1\ \sigma_{a_1}) \ldots (m_n\ \sigma_{a_n})$, where $\forall i.\sigma_i \notin FV(f)$ and $\forall j.m_j \in \Sigma_c(\sigma_{a_j})$.*
- *e satisfies the integrity policy il, if $\forall n_1 \in \Sigma_i(\sigma_1)$, ... $\forall n_k \in \Sigma_i(\sigma_k)$, for all f_{xy} that make the following substitutions well-typed,*

  $$[(n_1\ f_{11}\ f_{12}\ ...)/\sigma_1] \ldots [(n_k\ f_{k1}\ f_{k2}\ ...)/\sigma_k]\ e \equiv \begin{pmatrix} \mathsf{if}\ e_1(m_1\ e_{11}\ e_{12}\ ...) \\ \mathsf{if}\ e_2(m_2\ e_{21}\ e_{22}\ ...) \\ .... \\ \mathsf{if}\ e_n(m_n\ e_{n1}\ e_{n2}\ ...) \\ \qquad (m_0\ e_{01}\ e_{02}\ ...) \end{pmatrix}$$

  where $\forall i.\sigma_i \notin FV(f_{xy})$ and $\forall j.m_j \in il$.

In Definition 3.4.2, security policies are uniformly specified on both ends the program: Σ_c, Σ_i specify policies on the program inputs and cl, il specify policies on the program output. The confidentiality condition allows the program output to have security levels other than public. Compared to the integrity condition in Definition 3.4.1 where the program is simply untrusted, Definition 3.4.2 allows us to give trusted data to an untrusted program yet still having guarantees on the program output. The integrity condition looks more verbose because we have to use a lot of variables and term substitutions. However, the confidentiality condition and integrity condition are inherently symmetric except that the integrity condition allows implicit flows (via the if expressions).

3.5 Extensions

Similar to the idea of *global* downgrading policies in our previous framework [4], we can extend the policy language with secret variables. Although this significantly changes the confidentiality lattice (for example, the policy public is no longer the bottom of the lattice), the ordering of integrity labels is largely unchanged. In fact, doing so will only make the integrity policies more expressive. The integrity labels $\{\sigma_1\}$ and $\{c_1\}$ are very much alike — they are both singleton types; they are all lowest labels in the integrity ordering.

4 Conclusion

This paper studies the challenges on integrity policies in language-based information-flow security and provides a symmetrical view of confidentiality and integrity in the framework of *downgrading policies*. Although it is a common belief that confidentiality and integrity are duals, there are many aspects where integrity policies are fundamentally different from confidentiality policies. Integrity policies should precisely describe the computations on data in addition to the sources of data. The traditional noninterference-based approach provides no integrity guarantees for untrusted code, and is often too strong when dealing with implicit information flow.

This paper extended the framework of *downgrading policies* by presenting an more expressive model of integrity policies, where each label describe a set of possible functions that could have computed the data in the past. The presentations of confidentiality policies and integrity policies are mostly symmetrical. Traditional security levels for information-flow integrity such as tainted and untainted can be elegantly interpreted in this framework.

The asymmetry between confidentiality and integrity is shown in the ordering of security labels and also in the formalization of security goals. The interpretation and the ordering of integrity labels show the reason that untainted provides a weak security guarantee and suggests the use of more precise integrity labels instead of untainted. The definition of the security goal for integrity permits information leak through control flow yet provides formal, intuitive and practically useful security guarantees.

References

[1] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[2] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[3] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, September 2003.

[4] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 158–170, January 2005.

[5] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[6] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[7] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[8] Charles P. Pfleeger. *Security in Computing*, pages 5–6. Prentice-Hall, 1997. Second Edition.

[9] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, September 2000.

[10] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.

[11] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[12] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[13] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

Keeping Secrets in Incomplete Databases [Extended Abstract*]

Joachim Biskup and Torben Weibert
University of Dortmund, D-44221 Dortmund, Germany
{biskup|weibert}@ls6.cs.uni-dortmund.de

May 15, 2005

Abstract

Controlled query evaluation preserves confidentiality in information systems at runtime. A security policy specifies the facts a certain user is not allowed to know. At each query, a censor checks whether the answer would enable the user to learn any classified information. In that case, the answer is distorted, either by lying or by refusal. We introduce a framework in which controlled query evaluation can be analyzed wrt. possibly incomplete logic databases. For each distortion method – lying and refusal – a class of confidentiality-preserving mechanisms is presented. Furthermore, we specify a third approach that combines lying and refusal and compensates the disadvantages of the respective uniform methods.
Keywords: Information systems; Incomplete databases; Controlled query evaluation; Inference control.

1 Introduction

One basic requirement of a secure information system is preservation of *confidentiality*: Each piece of information may only be accessed by people who are authorized to do so. This is often enforced by the means of *static access rights* assigned to the structures of the information system. Although widely used, this approach entails a number of problems, one of which is the *inference problem*: The user might combine data he has access to in order to infer information he is not allowed to know. The inference problem has been studied in various contexts, for example statistical (see [10, 11, 13] for an introduction and e. g. [18, 19] for more recent work), multi-level and relational databases (see e. g. [9, 10, 15, 16, 20]). See [12] for a comprehensive review of the various approaches.

Controlled query evaluation is a dynamic approach to ensuring confidentiality, based on a fundamental logical framework. A *security policy* specifies the facts a certain user is not allow to learn. Then, at each query, it is checked whether the answer to that query would allow the user to infer any sensitive information. If this is the case, the answer is *distorted* by either *lying* or *refusal*. Though computationally expensive, this approach guarantees that no information will flow through otherwise unidentified inference channels. Controlled query evaluation was first proposed by Sicherman et al. [17] (presenting refusal as a distortion method) and Bonatti et al. [8] (introducing lying). Biskup [2] presents a unified framework for complete information systems in which both lying and refusal can be studied. Further work by Biskup and Bonatti relies on this framework and studies various aspects, including a comparison of lying and refusal [4] and a combined lying and refusal approach [5, 6]. A complete survey of the methods for complete databases can be found in [3].

Most information systems encountered in the real world are *incomplete*, in the respect that they cannot provide an answer to each query, as they have only limited knowledge. The goal of this paper is to adapt part of the existing work on controlled query evaluation to (possibly) incomplete logic databases. In our framework, a database instance db is a consistent set of propositional sentences. A query Φ, which is a propositional sentence, is either

*A draft of the complete paper containing detailed formalizations and proofs is available from the authors on request.

true, *false* or *undef* in db depending on whether Φ, its negation $\neg\Phi$ or neither of these are logically implied by db. The security policy is specified by a set of *potential secrets*, each of which is a propositional sentence. Before a query result is passed to the user, a *censor* checks whether this information would – now or later – enable the user to infer any of the potential secrets. In that case, the answer is distorted. When lying, a value different from the actual query value is provided as the answer, for example *false* instead of *true*. When refusing, the special answer *refuse* is returned. We present three different classes of censors: One that exclusively uses lying as a distortion method, one that uses only refusal, and a combined one that exploits both lying and refusal.

We say that a method of controlled query evaluation preserves *confidentiality* iff for each database instance, security policy, potential secret and query sequence, there is always a (possibly different) database instance in which that potential secret is not *true*, and under which the same answers would have been returned. Thus, the user cannot rule out that this potential secret is not *true*.

An important property of controlled query evaluation is that it keeps track of the information disclosed by earlier queries in order to avoid harmful inferences. This is done by the means of the *user log*. Prior to the first query, the user log contains the assumed *initial knowledge* of the user. Then, after each query, the information provided to the user is stored in the user log. Within the user log, we use *epistemic logic* in order to formalize the information already disclosed. For the uniform lying method and for the combined lying and refusal method, it is sufficient to store what answers the system provided to the queries. For the uniform refusal method, *meta inferences* evolve as a major problem: Having knowledge about the algorithm of the censor, the user can infer which query values might have led to a refusal in a certain situation. That way, he can draw inferences about the query value despite the fact that the answer has been refused. The refusal method overcomes this problem by storing all possible meta inferences from the answers in the user log.

The remainder of this paper is organized as follows: In Section 2, we give an overview about the basic ideas behind controlled query evaluation and demonstrate the functioning of a censor. In Section 3, we outline the formalization of our framework. The three enforcement methods – uniform lying, uniform refusal, and combined lying and refusal – and their properties are sketched in Section 4. Finally, we conclude in Section 5.

2 Controlled Query Evaluation

In this section, we present the basic concept of controlled query evaluation. First, we define the notion of incomplete logic databases, and give a definition of the security policy. Then, on a rather informal level, we demonstrate the functioning of a censor. The formalization of these ideas is then given in Section 3.

Ordinary Query Evaluation We consider (possibly) incomplete logic databases based on propositional logic. A database schema DS is a finite set of propositions. A database instance db over the schema DS is a consistent set of sentences, using only propositions from DS. The user issues a finite sequence of queries $Q = \langle \Phi_1, \ldots, \Phi_n \rangle$, each of which is a propositional sentence. The query Φ_i is either *true*, *false* or *undef* in db, depending on whether Φ_i, its negation $\neg\Phi_i$, or neither Φ_i nor $\neg\Phi_i$ are logically implied by db (as db is consistent, exactly one possibility holds). This is formalized as the following function ($\models_{PL}$ denotes logical implication in propositional logic):

$$
\begin{aligned}
eval(\Phi)(db) := \; & \texttt{if } db \models_{PL} \Phi \texttt{ then } \textit{true} \\
& \texttt{else if } db \models_{PL} \neg\Phi \texttt{ then } \textit{false} \\
& \texttt{else } \textit{undef}
\end{aligned}
$$

The Security Policy The security policy defines the facts to be hidden from the user. It consists of a set $pot\_sec = \{\Psi_1, \ldots, \Psi_m\}$ of propositional sentences, so-called *potential secrets*. The semantics is as follows: If a potential secret Ψ is *true* in the given database instance db, the user is not allowed to infer this information.

Security Configuration C	$eval(\Phi)(db) = ...$ *true*	*false*	*undef*
$\{\{true\}, \{false\}, \{undef\}\}$	*refuse*	*refuse*	*refuse*
$\{\{true\}, \{false\}\}$	*undef*	*undef*	*undef*
$\{\{true\}, \{undef\}\}$	*false*	*false*	*false*
$\{\{false\}, \{undef\}\}$	*true*	*true*	*true*
$\{\{true\}\}$	*undef*	*false*	*undef*
$\{\{false\}\}$	*true*	*undef*	*undef*
$\{\{undef\}\}$	*true*	*false*	*false*
$\emptyset$	*true*	*false*	*undef*

Table 1: A combined lying and refusal censor

On the other hand, if Ψ is *false* or *undef*, this information may be disclosed. This is a suitable formalization for real-life situations where the circumstance *that* a certain fact is true must be kept secret, but not the converse. For example, imagine a person applying for an employment. If that person suffered from a terminal disease, this must be kept secret (as it might be an obstacle for being chosen for the job). On the other hand, if the applicant is healthy, this information may be disclosed. For complete information systems, another type of security policies with different semantics, called *secrecies*, has been studied as well (see e. g. [3]), but in this paper, we concentrate on potential secrets. Additionally, we assume that the user *knows* the set of potential secrets, but of course not their respective values in the given database instance.

Our formal definition of confidentiality is given in Section 3. In a nutshell, it can be summarized as follows: Whatever the actual database instance db_1 and the security policy *pot_sec* are, and whatever sequence of queries Q the user issues, there must be a database instance db_2, in which a given potential secret Ψ is not *true*, and under which the same answers would have been returned. Then, the user cannot decide from the answers whether db_1 or db_2 is the actual database instance, and thus whether Ψ is *true* or not.

Note that this is a rather declarative definition, and it does not include any hint about which techniques to be used to operationally meet these requirements. The approach described below keeps a log of the information already disclosed to the user, and each piece of information in that log is formalized as a sentence in epistemic logic. Nevertheless, modal logic is only exploited as an auxiliary means here, but not to describe the situation in general.

The Censor The most important component of our framework is the *censor function* that decides whether an answer needs to be distorted in order to preserve confidentiality, and, if so, in which manner. The censor function takes two parameters: 1. The actual value of the query $eval(\Phi)(db)$, and 2. the so-called *security configuration* C, describing the threats to the security policy in the given situation. The result of the censor function is the answer to be returned to the user, either *true*, *false* or *undef* (which might differ from the actual query value, in case lying is used as a distortion technique), or even the special answer *refuse*, indicating that the answer is refused.

A censor function can easily be written as a decision table, with each line representing a security configuration and each column representing a query value. Table 1 gives an example of a censor using both lying and refusal as a distortion method. The black cells denote the situations in which a distorted answer is given. The policy of this censor is to use lying as a distortion method whenever necessary and possible, and use refusal if lying would fail.

We formalize each piece of information disclosed to the user as an *inference set*, i. e., as a set of values the user regards as possible wrt. the actual query value. Unary inference sets describe complete information about a query value. For example, $\{true\}$ denotes that the user learns that the query value is *true*. The uniform refusal censor will also have to consider binary inference sets, describing incomplete information about the query value.

A security configuration is given as a set $C = \{V_1, \ldots, V_k\}$ of inference sets, all of which would lead to a

disclosure of sensitive information. For example, the security configuration $C = \{\{\mathit{true}\}, \{\mathit{false}\}\}$, as in the second line of the censor table, denotes that neither the answer *false* nor the answer *true* may be given, as both would lead to a violation of the security policy. In Section 3, we show how the security configuration is determined based on the *user log*, which contains information about all facts disclosed to the user so far.

Having understood the concept of security configurations, one can easily see how the censor works. For example, when neither *true* nor *false* may be given as an answer (second line of Table 1), the censor responds to this situation by returning the remaining safe answer *undef*, even if the actual query value is *true* or *false*. For the three security configurations $\{\{\mathit{true}\}\}$, $\{\{\mathit{false}\}\}$ and $\{\{\mathit{undef}\}\}$, the censor has two safe answers to choose from. The security configuration $\{\{\mathit{true}\}, \{\mathit{false}\}, \{\mathit{undef}\}\}$ in the first line of the censor table represents the special situation in which none of the values *true*, *false* and *undef* may be given as an answer, as all of them would lead to a security violation. In that case, the censor refuses to answer, as a last resort. Finally, the security configuration $\emptyset$ in the last line means that neither answer is dangerous, allowing the censor to pass the unmodified query value to the user.

A special problem arises when we disallow refusal as a distortion method. As you have seen, the security configuration $\{\{\mathit{true}\}, \{\mathit{false}\}, \{\mathit{undef}\}\}$ represents the situation where all of the values *true*, *false* and *undef* may not be given as an answer, as all of them would allow the user to infer sensitive information. While the combined lying and refusal censor overcomes this situation by refusing to answer, the uniform lying mechanism has to make certain arrangements so that this "hopeless situation" will not occur. This is done by modifying the notion of a violation of the security policy, protecting the disjunction of all potential secrets instead of each single one, as described in Section 3.

Refusal and Meta Inferences A first approach to refusal in incomplete databases is given in [7]. In the present work, we adapt the ideas from that paper to our new framework and give a formal definition of the *meta inferences* that evolve as the major problem when refusal is used.

Meta inferences occur when the user combines the answers received by the system with his knowledge about how the system works. The latter is made up of two parts: 1. We follow the principle of open design and assume that the user knows the algorithm of the censor and its auxiliary components. 2. The security configuration is computed only by information the user has access to (the user log, the query, and, as we assume, the security policy). Thus, whenever the user receives an answer from the system, he can determine what line of the censor table this answer originates from, and compare the received answer to the answers found in that line.

For example, consider the following excerpt from a censor table:

C	*true*	*false*	*undef*
$\{\{\mathit{true}\}, \{\mathit{false}\}\}$	*refuse*	*refuse*	*undef*

Imagine the user issues a query Φ, which has the value $eval(\Phi)(db) = \mathit{true}$ and the associated security configuration $C = \{\{\mathit{true}\}, \{\mathit{false}\}\}$, meaning that both answering *true* and *false* would lead to a security violation. Thus, the censor answers *refuse*. The user can now determine the security configuration and finds that *refuse* is only returned as an answer if the query value is either *true* or *false*, but not *undef*. So the user has gained partial information about the query value, which we express by a binary inference set, in particular $\{\mathit{true}, \mathit{false}\}$. Even this partial information might be harmful (as a matter of fact, if both the answers *true* and *false* would lead to the disclosure of the same potential secret, the partial information "the value is either *true* or *false*" is sufficient to infer that secret).

In order to overcome this problem, the uniform refusal method takes two actions: 1. Also binary inference sets are considered when determining the security configurations, i. e., the inferences harmful to the security policy (which, admittedly, leads to computational overhead, as a greater number of implications need to be computed). 2. In case a harmful meta inference can be drawn from an answer, an additional *refuse* is introduced in that line of the censor table, avoiding the respective meta inference (which on the other hands leads to a lack of availability, as more answers need to be refused). For example, the above mentioned situation can be solved as follows:

C	*true*	*false*	*undef*
$\{\{true\}, \{false\}\}$	*refuse*	*refuse*	*undef*
$\{\{true\}, \{false\}, \{true, false\}\}$	*refuse*	*refuse*	*refuse*

The first line represents the situation where both the inferences $\{true\}$ and $\{false\}$ would lead to a security violation, but the partial inference $\{true, false\}$ would not. In that case, it is sufficient to refuse the answer only if the query value is *true* or *false*. The second line represents the situation where also the partial information is harmful. Introducing an additional *refuse* in the third column (for the value *undef*), the user cannot draw that harmful meta inference anymore (as any value would have led to the answer *refuse*).

The uniform lying method does not need to consider meta inferences because the mechanism of lying, as used by our censors, avoids them in the first place by choosing only *harmless* answers as a lie. Consider the following example:

C	*true*	*false*	*undef*
$\{\{true\}\}$	*false*	*false*	*undef*

Under this security configuration, *true* is a harmful answer, so it is replaced by the harmless answer *false*. The resulting meta inference from the answer *false* is then $\{true, false\}$. This partial inference is harmless, because even the more precise inference $\{false\}$ is. (If it is harmless to know that the value is *false*, it is also harmless to know that the value is either *true* or *false*). By choosing only harmless answers as a lie, the censor guarantees that the meta inference includes at least one harmless component, making the whole meta inference harmless as well.

The combined lying and refusal method, which is derived from the uniform lying method, is not affected by meta inferences as well; on the one hand, it inherits the abovementioned properties of the uniform lying method, and on the other hand, under the additionally introduced security configuration $\{\{true\}, \{false\}, \{undef\}\}$, the answer is refused regardless of the query value, eliminating harmful meta inferences as well.

Storing the actual answers (as performed by the uniform lying and the combined lying and refusal method) instead of the meta inferences (as used by the uniform refusal method) is advantageous for two additional reasons: First, it leads to an increase in computational efficiency, as a lower number of inferences has to be checked for a violation of the security policy. Second, the censor is able to keep the set of answers consistent, even if there are lies among them.

3 Formalization

Having introduced the basic ideas of controlled query evaluation, we now outline a formal definition of its various components.

The User Log Controlled query evaluation keeps track of the facts already disclosed to the user in order to determine which inferences would allow the user to infer any of the potential secrets. This information is stored in the *user log*. Prior to the first query, we have the assumed initial user knowledge log_0. Then, after each query Φ_i, the inference from the ith answer ans_i is added to the user log. In order to formalize the inferences, we use *epistemic logic* (S5), established by introducing the modal operator K which is to be read as "it holds in the database that ...". A query Φ and the inference set $\emptyset \neq V \subseteq \{true, false, undef\}$ from the respective answer ans can then be converted into a sentence of epistemic logic by the function $\Delta^*(\Phi, V) = \bigvee_{v \in V} \Delta(\Phi, v)$ with

$$\Delta(\Phi, true) = K\Phi, \;\; \Delta(\Phi, false) = K\neg\Phi, \;\; \Delta(\Phi, undef) = \neg K\Phi \wedge \neg K\neg\Phi.$$

We assume that each enforcement method defines a function $inference(censor, C, ans)$ that computes the inference from an answer ans (which was given by the censor *censor* under the security configuration C). As

mentioned in Section 2, we use two different approaches: The *uniform lying* method and the *combined lying and refusal* method take the answer as the inference set (while refusals are discarded), formalized by the function

$$\textit{inference}^{ans}(censor, C, ans) \;:=\; \texttt{if}\ ans = \textit{refuse}\ \texttt{then}\ \emptyset\ \texttt{else}\ \{ans\}.$$

On the other hand, the *uniform refusal* method considers the meta inferences a highly-sophisticated user can draw from an answer. The meta inference corresponds to the set of values that lead to the answer ans under the given security configuration C, in formulae:

$$\textit{inference}^{meta}(censor, C, ans) \;=\; \{\ v \mid v \in \{\textit{true}, \textit{false}, \textit{undef}\} \text{ and } censor(C, v) = ans\ \}.$$

The range of these functions, i. e., the set of inferences that can occur under a certain enforcement method, is given by $\mathcal{I}^{ans} = \{\{\textit{true}\}, \{\textit{false}\}, \{\textit{undef}\}, \emptyset\}$ and $\mathcal{I}^{meta} = \mathfrak{P}^{+}(\{\textit{true}, \textit{false}, \textit{undef}\})$ respectively (where $\mathfrak{P}$ is the power set operator and $^{+}$ indicates that the empty set is excluded).

Security Violations In order to prevent a violation of the security policy, we need to define what it means that such a violation is existent. Again, we assume that each enforcement method provides a function *violates*$(\textit{pot\_sec}, log)$, deciding whether the user log enables the user to learn any of the potential secrets. It is guaranteed as a precondition that the initial user log log_0 does not violate the security policy. Later, this is kept as an invariant.

We use two different versions of this function. The *uniform refusal* method and the *combined* method define that there is a violation if one of the potential secrets from *pot_sec* is logically implied (wrt. epistemic logic) by the user log:

$$\textit{violates}^{single}(\textit{pot\_sec}, log) \;:=\; (\exists \Psi \in \textit{pot\_sec})[log \models_{S5} \Psi].$$

On the other hand, the *uniform lying* method needs to avoid the hopeless situation in which neither answer may be given. This is achieved by the stricter *violates*-function

$$\textit{violates}^{disj}(\textit{pot\_sec}, log) \;:=\; log \models_{S5} \bigvee_{\Psi \in \textit{pot\_sec}} \Psi.$$

For example, consider the security policy $\textit{pot\_sec} = \{s_1, s_2, s_3\}$ and the user log $log = \{Ka \rightarrow s_1, K\neg a \rightarrow s_2, \neg Ka \wedge \neg K\neg a \rightarrow s_3\}$. According to $\textit{violates}^{single}$, this log does not violate the security policy (as the user does now know *which* of the potential secrets holds). On the other hand, this user log does violate the security policy according to $\textit{violates}^{disj}$ (as the user knows that at least one of the potential secrets holds). Moreover, when the user issues the query $\Phi = a$ and refusal is not allowed as a distortion method, we have the aforementioned "hopeless situation" in which either answer would disclose a potential secret.

The uniform lying method avoids this situation by keeping the stricter condition $\textit{violates}^{disj}$ as an invariant for all user logs generated throughout the query sequence. On the other hand, the refusal and the combined method adhere to the weaker condition $\textit{violates}^{single}$, for the sake of availability.

Security Configurations As outlined in Section 2, the security configuration of a query is the set of inferences that would lead to a user log violating the security policy. This can be formalized as the function

$$\textit{sec\_conf}(\textit{pot\_sec}, log, \Phi) = \{\ V \mid V \in \mathcal{I} \text{ and } \textit{violates}(\textit{pot\_sec}, log \cup \{\Delta^{*}(\Phi, V)\})\}$$

where *pot_sec* is the security policy, log the current user log, Φ the query and $\mathcal{I}$ the range of the version of the function *inference* used by this method. Note that the user is assumed to know all of these parameters, so he can himself calculate the security configuration, enabling him to draw meta inferences. The range of *sec_conf* is limited by $\mathcal{I}$ and certain constraints. For example, an inference $\{v_1, v_2\}$ can only be dangerous if both $\{v_1\}$ and $\{v_2\}$ are. The censors will only have to handle those relevant security configurations.

The Censor The censor is the most important component of controlled query evaluation. It considers the security configuration (i. e., the current threats to the security policy) and the actual query value and decides what answer to give, either the original query value or a modified answer. This can be formalized as the function

$$censor : \mathfrak{P}(\mathfrak{P}^+(\{\mathit{true}, \mathit{false}, \mathit{undef}\})) \times \{\mathit{true}, \mathit{false}, \mathit{undef}\} \to \{\mathit{true}, \mathit{false}, \mathit{undef}, \mathit{refuse}\}.$$

For each enforcement method, we will restrict the censor function to certain behavior according that method (lying, refusal or combined) and then state a couple of requirements such a censor must meet in order to preserve confidentiality.

Formalization and Security Based on the components described above, we can define a method of controlled query evaluation as a function

$$control\_eval(Q, log_0, db, pot\_sec) := \langle (ans_1, log_1), \ldots, (ans_n, log_n) \rangle$$

where $Q = \langle \Phi_1, \ldots, \Phi_n \rangle$ is a query sequence, log_0 the initial user log, db a database instance, and pot_sec a set of potential secrets. In each step i, the answer ans_i and the subsequent user log log_i are generated as follows:

1. Determine the security configuration: $C_i = \mathit{sec\_conf}(pot\_sec, log_{i-1}, \Phi_i)$
2. Let the censor generate the answer: $ans_i = \mathit{censor}(C_i, eval(\Phi_i)(db))$
3. Add the corresponding inference to the user log: $log_i = log_{i-1} \cup \{\Delta^*(\Phi_i, \mathit{inference}(censor, C_i, ans_i))\}$

Each method goes along with a function *precondition* that defines the admissible arguments for that method. For the methods described in this work, it is demanded that the initial user log does not violate the security policy:

$$\mathit{precondition}(db, log_0, \mathit{pot\_sec}) := \texttt{not } \mathit{violates}(\mathit{pot\_sec}, log_0)$$

Our notion of confidentiality can then be defined as follows: Let $control\_eval$ be a controlled query evaluation with $precondition$ as associated precondition for admissible arguments. $control\_eval$ is defined to preserve confidentiality iff

> for all finite query sequences Q, for all security policies *pot_sec*, for all potential secrets $\Psi \in pot\_sec$, for all initial user logs log_0, for all instances db_1 so that (db_1, log_0, pot_sec) satisfies *precondition*, there exists db_2 so that (db_2, log_0, pot_sec) satisfies *precondition*, and the following conditions hold:
>
> (a) [db_1 and db_2 produce the same answers]
> $control\_eval(Q, log_0, db_1, pot\_sec) = control\_eval(Q, log_0, db_2, pot\_sec)$
>
> (b) [Ψ is not *true* in db_2]
> $eval(\Psi)(db_2) \in \{\mathit{false}, \mathit{undef}\}$

4 Enforcement Methods

In the following, we give a brief overview of the enforcement methods for controlled query evaluation suitable for incomplete databases. Each method is identified by its basic policy (lying, refusal or both), and the *inference*-function and the *violates*-function it uses. Furthermore, we state some requirements the censor function must meet for each security configuration C occuring under that method in order to preserve confidentiality. There are theorems stating that a censor meeting these requirements preserves confidentiality as defined in Section 3. Examples of the three types of censors can be found in Tables 1, 2 and 3.

Lying

Censor policy	only lie, never refuse
Inferences	$\mathit{inference}^{ans}$ (inference corresponds to answer)
Security violations	$\mathit{violates}^{disj}$ (disjunction of all potential secrets)
Censor requirements	for each $v \in \{\mathit{true}, \mathit{false}, \mathit{undef}\}$: (a) [safe answers] $\{censor(C, v)\} \notin C$ (b) [only lie if necessary] if $\{v\} \notin C$ then $censor(C, v) = v$

Refusal

Censor policy	only refuse, never lie
Inferences	$\mathit{inference}^{meta}$ (meta inferences)
Security violations	$\mathit{violates}^{single}$ (single potential secrets)
Censor requirements	meta inference may not lead to a violation: for each $ans \in \{\mathit{true}, \mathit{false}, \mathit{undef}, \mathit{refuse}\}$: $\mathit{inference}^{meta}(censor, C, ans) \notin C$

Combined lying and refusal

Censor policy	lie as long as possible, otherwise refuse
Inferences	$\mathit{inference}^{ans}$ (inference corresponds to answer)
Security violations	$\mathit{violates}^{single}$ (single potential secrets)
Censor requirements	(a) [safe answers] for each $v \in \{\mathit{true}, \mathit{false}, \mathit{undef}\}$: $\{\mathit{censor}(C, v)\} \notin C$ (b) [only lie if necessary] for each $v \in \{\mathit{true}, \mathit{false}, \mathit{undef}\}$: if $\{v\} \notin C$ then $censor(C, v) = v$ (c) [no meta inferences from refusals] if $censor(C, v) = \mathit{refuse}$ for any $v \in \{\mathit{true}, \mathit{false}, \mathit{undef}\}$, then $censor(C, v) = \mathit{refuse}$ for all $v \in \{\mathit{true}, \mathit{false}, \mathit{undef}\}$

Security Configuration C	$eval(\Phi)(db) = ...$ *true*	*false*	*undef*
$\{\{\mathit{true}\}, \{\mathit{false}\}\}$	*undef*	*undef*	*undef*
$\{\{\mathit{true}\}, \{\mathit{undef}\}\}$	*false*	*false*	*false*
$\{\{\mathit{false}\}, \{\mathit{undef}\}\}$	*true*	*true*	*true*
$\{\{\mathit{true}\}\}$	*undef*	*false*	*undef*
$\{\{\mathit{false}\}\}$	*true*	*undef*	*undef*
$\{\{\mathit{undef}\}\}$	*true*	*false*	*false*
$\emptyset$	*true*	*false*	*undef*

Table 2: A lying censor

The proof idea is similar to all three types of censors: The censors guarantee that the final user log log_n does not logically imply any potential secret $\Psi \in \mathit{pot\_sec}$. Thus, there must be an S5-structure M and a state s so that $(M, s) \models log_n$ and $(M, s) \not\models \Psi$. Defining db_2 as the set of all propositional sentences ϕ so that $(M, s) \models K\phi$, we have $eval(\Psi)(db_2) = \mathit{false}$. It can be shown by induction that db_2 also returns the same answers as db_1.

It turns out that the combined lying and refusal method has advantages over the two uniform methods, for three reasons: 1. Unlike the uniform lying method, it protects each single potential secret but not the disjunction of all secrets, leading to a gain of availability. 2. Unlike the uniform refusal method, it does not consider partial

Security Configuration C	$eval(\Phi)(db) = ...$ *true*	*false*	*undef*
$\{\{true\}, \{false\}, \{undef\}, ...\}$	*refuse*	*refuse*	*refuse*
$\{\{true\}, \{false\}, \{true, false\}\}$	*refuse*	*refuse*	*refuse*
$\{\{true\}, \{false\}\}$	*refuse*	*refuse*	*undef*
$\{\{true\}, \{undef\}, \{true, undef\}\}$	*refuse*	*refuse*	*refuse*
$\{\{true\}, \{undef\}\}$	*refuse*	*false*	*refuse*
$\{\{true\}\}$	*refuse*	*false*	*refuse*
$\{\{false\}, \{undef\}, \{false, undef\}\}$	*refuse*	*refuse*	*refuse*
$\{\{false\}, \{undef\}\}$	*true*	*refuse*	*refuse*
$\{\{false\}\}$	*true*	*refuse*	*refuse*
$\{\{undef\}\}$	*true*	*refuse*	*refuse*
$\emptyset$	*true*	*false*	*undef*

Table 3: A refusal censor. The first line is an abbreviation for all security configurations which are a superset of $\{\{true\}, \{false\}, \{undef\}\}$, all of which are treated in the same way by this censor.

information (i. e., binary inference sets), needing less computational power. 3. Unlike the uniform refusal method, it is not vulnerable to harmful meta inferences from refusals, and thereby needs no additional *refuse*-conditions, leading to a gain of availability.

5 Conclusion and Further Work

In the present paper, we have studied controlled query evaluation as a means to dynamically preserve confidentiality in databases that might be incomplete. We have presented three basic enforcement methods: Uniform lying, uniform refusal and combined lying and refusal. The insight that the combined method has advantages over the two uniform methods corresponds to the results for complete databases [6].

In [7], a refusal censor is presented that uses a mechanism similar to *inference*^{ans} in order to store inferences. While this is a saving of computational overhead, it leads to a lack of availability: That censor only checks unary inference sets for a violation, but not the binary ones. If two unary inference sets, for example, $\{true\}$ and $\{false\}$, are identified as harmful, it is, as a precaution, assumed that also their union $\{true, false\}$ is. So an additional *refuse*-condition for $eval(\Phi)(db) =$ *undef* is introduced in order to avoid that harmful meta inference. In this respect, the censor from Table 3 is more cooperative, as it explicitly checks whether $\{true, false\}$ is harmful, and only introduces the additional *refuse* if that is the case. Obviously, we have a trade-off between availability and complexity: A refusal censor that checks fewer inferences for a possible violation avoids computational overhead but has to refuse the answer more often as a precaution, leading to a lack of availability. For example, a censor that always refuses to answer is very fast but leads to total unavailability. On the other hand, the censor from this paper was constructed by adding exactly one additional *refuse*-condition at each of the six security configurations affected by harmful meta inferences. This heuristics is intended to guarantee maximum availability, though we have no formal proof for this proposition.

So far, our work is limited to propositional logic. At first glance, this is a serious restriction. Nevertheless, at this time, our framework may already be integrated into simple applications that involve only a small set of atoms. Such an integration will be the subject of future work. More complex applications demand that higher logics are used, for example first-order logic. Then, implication is only decidable within certain restrictions, for example under a fixed finite domain, or when only certain types of sentences are allowed. We also get decidability if we consider finite implications, i. e., taking care of finite models only [14]. It is still to be fully analyzed which

versions or fragments of relational databases fit our framework, in particular when allowing open queries rather than only closed (yes/no-)queries as in the present paper (see for example [1] for an overview about complexity and decidability issues in relational databases).

Furthermore, we assume that exactly one user is querying the database. When we have a set of different, possibly colluding users, there are two options: First, one can regard all users as being the same and sharing the same user log. Second, one can keep a separate user log for each of the users. The latter option is the more interesting one, though it demands that there is a formal representation of the collusions, i. e., of the partial information the users share with each other.

Finally, there are some aspects already studied for complete information systems [3] which still have to be adapted to our new framework, including different kinds of security policies and the special case that the user does not know the elements of the security policy, which might be exploited by our system in order to achieve a gain of availability.

References

[1] Serge Abitehoul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Joachim Biskup. For unknown secrecies refusal is better than lying. *Data & Knowledge Engineering*, 33:1–23, 2000.

[3] Joachim Biskup and Piero Bonatti. Controlled query evaluation for enforcing confidentiality in complete information systems. *International Journal of Information Security*, 3:14–27, 2004.

[4] Joachim Biskup and Piero A. Bonatti. Lying versus refusal for known potential secrets. *Data & Knowledge Engineering*, 38:199–222, 2001.

[5] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. In Thomas Eiter and Klaus-Dieter Schewe, editors, *FoIKS*, volume 2284 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2002.

[6] Joachim Biskup and Piero A. Bonatti. Controlled query evaluation for known policies by combining lying and refusal. *Annals of Mathematics and Artificial Intelligence*, 40:37–62, 2004.

[7] Joachim Biskup and Torben Weibert. Refusal in incomplete databases. In Csilla Farkas and Pierangela Samarati, editors, *Research Directions in Data and Applications Security XVIII*, volume 144, pages 143–157. Kluwer/Springer, 2004.

[8] Piero A. Bonatti, Sarit Kraus, and V.S. Subrahmanian. Foundations of secure deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):406–422, 1995.

[9] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Trans. Knowl. Data Eng.*, 12(6):900–919, 2000.

[10] Silvano Castano, Mariagrazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. ACM Press, 1995.

[11] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[12] Csilla Farkas and Sushil Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–11, 2002.

[13] Ernst L. Leiss. *Principles of Data Security*. Plenum Press, 1982.

[14] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[15] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman, and William R. Shockley. The seaview security model. *IEEE Trans. Software Eng.*, 16(6):593–607, 1990.

[16] Xiaolei Qian and Teresa F. Lunt. A semantic framework of the multilevel secure relational model. *IEEE Trans. Knowl. Data Eng.*, 9(2):292–301, 1997.

[17] George L. Sicherman, Wiebren de Jonge, and Reind P. van de Riet. Answering queries without revealing secrets. *ACM Transactions on Database Systems*, 8(1):41–59, 1983.

[18] Lingyu Wang, Sushil Jajodia, and Duminda Wijesekera. Securing OLAP data cubes against privacy breaches. In *IEEE Symposium on Security and Privacy*, pages 161–178. IEEE Computer Society, 2004.

[19] Lingyu Wang, Yingjiu Li, Duminda Wijesekera, and Sushil Jajodia. Precisely answering multi-dimensional range queries without privacy breaches. In Einar Snekkenes and Dieter Gollmann, editors, *ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2003.

[20] Marianne Winslett, Kenneth Smith, and Xiaolei Qian. Formal query languages for secure relational databases. *ACM Trans. Database Syst.*, 19(4):626–662, 1994.

Non-Interference for a Typed Assembly Language

Ricardo Medel Adriana Compagnoni Eduardo Bonelli
Stevens Institute of Technology LIFIA, Fac. de Informática, UNLP
Hoboken, NJ La Plata (Argentina)

Abstract

Non-interference is a desirable property of systems in a multilevel security architecture, stating that confidential information is not disclosed in public output. The challenge of studying information flow for assembly languages is that the control flow constructs that guide the analysis in high-level languages are not present. To address this problem, we define a typed assembly language that uses pseudo-instructions to impose a stack discipline on the control flow of programs. We develop a type system for checking that assembly programs enjoy non-interference and its proof of soundness.

1 Introduction

The confidentiality of information handled by computing systems is of paramount importance. However, standard perimeter security mechanisms such as access control or digital signatures fail to address the enforcement of information-flow policies. On the other hand, language-based strategies offer a promising approach to information flow security. In this paper, we study confidentiality for an assembly language using a language-based approach to security via type-theory.

In a multilevel security architecture information can range from having low (public) to high (confidential) security level. Information flow analysis studies whether an attacker can obtain information about the confidential data by observing the output of the system. The non-interference property states that any two executions of the same program, where only the high-level inputs differ in both executions, does not exhibit any observable difference in the program's output.

In this paper we define SIF, a typed assembly language for secure information flow analysis with security types. This language contains two pseudo-instructions, `cpush` L and `cjmp` L, for handling a stack of code labels indicating the program points where different branches of code converge, and the type system enforces a stack policy on those code labels. Our development culminates with a proof that well-typed SIF programs are assembled to untyped machine code that satisfy non-interference.

The type system of SIF detects explicit illegal flows as well as implicit illegal flows arising from the control structure of a program. Other covert channels such as those based on termination, timing, and power consumption, are outside the scope of this paper.

2 SIF, A Typed Assembly Language

In information flow analysis, a security level is associated with the program counter (`pc`) at each program execution point. This security level is used to detect implicit information flow from high-level values to low-level values. Moreover, control flow analysis is crucial in allowing this security level to decrease where there is no risk of illicit flow of information.

Consider the example in Figure 1(a), where x has high security level and z has low security level. Notice that y cannot have low security level, since information about x can be retrieved from y, violating the non-interference property. Since the execution path depends on the value stored in the high-security variable x, entering the branches of the `if-then-else` changes the security level of the pc to high, indicating that only high-level variables can be updated. On the other hand, since z is modified after both branches, there is no leaking of information from either y or x to z. Therefore, the security level of the pc can be safely lowered.

Sec. level of pc	
low	`if x=0`
high	`then y:=1`
high	`else y:=2`
low	`z:=3`

(a) High-level program

```
L1 :  bnz r1, L2     % if x≠0 goto L2
      move r2 ← 1    % y:= 1
      jmp L3
L2 :  move r2 ← 2    % y:= 2
L3 :  move r3 ← 3    % z:= 3
```

(b) Assembly program

Figure 1: Example of implicit illegal information flow.

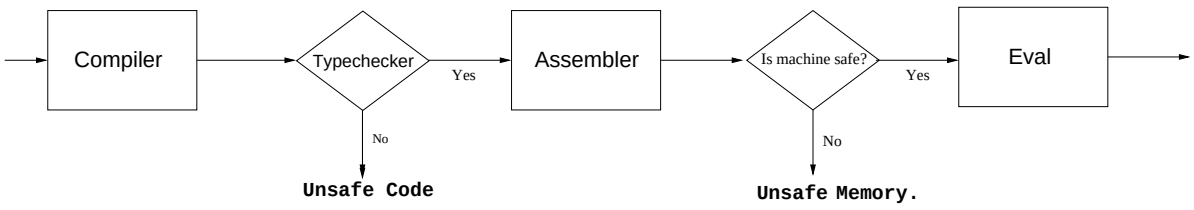
A standard compilation of this example to assembly language may produce the code shown in Figure 1(b). Note that the block structure of the `if-then-else` is lost, and it is not clear where it is safe to lower the security level of the pc. We address this problem by including in our assembly language a stack of code labels accessed by two pseudo-instructions, `cpush` L and `cjmp` L, to simulate the block structure of high-level languages.

The instruction `cpush` L pushes L onto the stack while `cjmp` L first pops L from the stack if L is already at the top, and then jumps to the instruction labelled by L. The extra label information in `cjmp` L allows us to statically control that the intended label is removed, thereby preventing ill structured code.

The SIF code for the example in Figure 1(a) is shown below. The code at $L1$ pushes the label $L3$ onto the stack. The code at $L3$ corresponds to the instructions following the `if-then-else` in the source code. Observe that the code at $L3$ can only be executed once, because the instruction `cjmp` $L3$ at the end of the code pointed to by $L1$ (`then` branch), or at the end of $L2$ (`else` branch), removes the top of the stack and jumps to the code pointed to by $L3$. At this point it is safe to lower the security level of the pc, since updating the low-security register r_3 does not leak any information about r_1.

```
L1 :  {r0 : int⊥, r1 : int⊤, r2 : int⊤, r3 : int⊥, pc : ⊥} ‖ ϵ
      cpush L3                          % set junction point L3
      bnz r1, L2                        % if x≠ 0 goto L2
      arithi r2 ← r0 + 1                % y:= 1, with r0=0
      cjmp L3
L2 :  {r0 : int⊥, r2 : int⊤, r3 : int⊥, pc : ⊤} ‖ L3 · ϵ
      arithi r2 ← r0 + 2                % y:= 2
      cjmp L3
L3 :  {r0 : int⊥, r3 : int⊥, pc : ⊥} ‖ ϵ
      arithi r3 ← r0 + 3                % z:= 3
      halt
      eof
```

Moreover, as in HBAL [1], the type-checking of the program is separated from the verification of the safety of the machine configuration where the program is assembled. Thus, following the schema shown below, a type-checker can verify if a program is safe for execution on *any safe* memory configuration, and the runtime environment only needs to check that the initial machine configuration is safe before each run.



The assembler removes `cpush` L and translates `cjmp` L into `jmp` L, an ordinary unconditional jump, leaving no trace of these pseudo-instructions in the executable code (see the definition of the assembly function $\mathsf{Asm}(-)$ in section 2.4).

2.1 The Type System

We assume given a lattice $\mathfrak{L}_{\mathbf{sec}}$ of *security labels* [8], with an ordering relation $\sqsubseteq$, least ($\bot$) and greatest ($\top$) elements, and join ($\sqcup$) and meet ($\sqcap$) operations. These labels assign security levels to elements of the language through types. The type expressions of SIF are given by the following grammar:

$$
\begin{array}{lllll}
\textit{security labels} & l & \in & \mathfrak{L}_{\mathbf{sec}} & \\
\textit{security types} & \sigma & ::= & \omega^l & \\
\textit{word types} & \omega & ::= & \mathit{int} \mid [\tau] & \\
\textit{memory location types} & \tau & ::= & \sigma \times \ldots \times \sigma \mid \texttt{code} &
\end{array}
$$

Security types (σ) are word types annotated with a security label. The expression $\textsc{Labl}(\sigma)$ returns the security label of a security type σ. A *word type* (ω) is either an integer type (int) or a pointer to a memory location type ($[\tau]$). *Memory location types* (τ) are tuples of security types, or a special type `code`. We use $\tau[c]$, with c a positive integer, to refer to the c^{th} word type of the product type τ. Since the type `code` indicates the type of an assembly instruction, our system distinguishes code from data.

A *context* ($\Gamma \parallel \Lambda$) contains a register context Γ and a junction points stack Λ. A *junction points stack* (Λ) is a stack of code labels, each representing the convergence point of a fork in the control flow of a program. The empty stack is denoted by ϵ. A *register context* Γ contains type information about registers, mapping them to security types. We assume a finite set of registers $\{r_0, \ldots, r_n\}$, with two dedicated registers: r_0, that always holds zero, and `pc`, the program counter.

We write $Dom(\Gamma)$ for the domain of the register context Γ. The empty context is denoted by $\{\}$. The register context obtained by eliminating from Γ all pairs with r as first component be denoted by $\Gamma_{/r}$, while Γ, Γ' denotes the union of register contexts with disjoint domains. We use $\Gamma, r : \sigma$ as a shorthand for $\Gamma, \{r : \sigma\}$, and $\Gamma[r := \sigma]$ as a shorthand for $\Gamma_{/r}, \{r : \sigma\}$.

Since the program counter is always a pointer to code, we usually write $\texttt{pc} : l$ instead of $\texttt{pc} : [\texttt{code}]^l$. We also use $\Gamma(\texttt{pc})$ instead of $\textsc{Labl}(\Gamma(\texttt{pc}))$.

2.2 Syntax of SIF programs

A *program* (P) is a sequence of instructions and code labels ended by the directive `eof`. SIF has standard assembly language instructions such as arithmetic operations, conditional branching, load, and store, plus pseudo-instructions `cpush` and `cjmp` to handle the stack of code labels.

$$
\begin{array}{lllll}
\textit{program} & P & ::= & \texttt{eof} \mid L; P \mid p; P & \\
\textit{instructions} & p & ::= & \texttt{halt} \mid \texttt{jmp}\ L \mid \texttt{bnz}\ r, L & \\
 & & \mid & \texttt{load}\ r \leftarrow r[c] \mid \texttt{store}\ r[c] \leftarrow r & \\
 & & \mid & \texttt{arith}\ r \leftarrow r \odot r \mid \texttt{arithi}\ r \leftarrow r \odot i & \\
 & & \mid & \texttt{cpush}\ L \mid \texttt{cjmp}\ L & \\
\textit{operations} & \odot & ::= & + \mid - \mid * \mid / &
\end{array}
$$

We use c to indicate an offset, and i to indicate integer literals. We assume an infinite enumerable set of code labels. Intuitively, the instruction `cpush` L pushes the junction point represented by the code label L onto the stack, while the instruction `cjmp` L behaves as a pop and a jump. If L is at the top of the stack, it pops L and then jumps to the instruction labeled L.

$$\frac{\Gamma' \subseteq \Gamma \quad l \sqsubseteq l'}{(\Gamma, \mathsf{pc} : l \parallel \Lambda) \leq (\Gamma', \mathsf{pc} : l' \parallel \Lambda)}\ \mathsf{ST\_RegBank} \qquad \frac{\mathsf{Ctxt}(P) \vdash_\Sigma P}{\Gamma \parallel \epsilon \vdash_\Sigma \texttt{halt}; P}\ \mathsf{T\_Halt} \qquad \frac{}{\Gamma \parallel \epsilon \vdash_\Sigma \texttt{eof}}\ \mathsf{T\_Eof}$$

$$\frac{(\Gamma \parallel \Lambda) \leq \Sigma(L) \quad \Sigma(L) \vdash_\Sigma P}{\Gamma \parallel \Lambda \vdash_\Sigma L; P}\ \mathsf{T\_Label} \qquad \frac{(\Gamma \parallel \Lambda) \leq \Sigma(L) \quad \mathsf{Ctxt}(P) \vdash_\Sigma P}{\Gamma \parallel \Lambda \vdash_\Sigma \texttt{jmp}\ L; P}\ \mathsf{T\_Jmp}$$

$$\frac{(\Gamma, r : int^{l'}, \mathsf{pc} : l \sqcup l' \parallel \Lambda) \leq \Sigma(L) \quad \Gamma, r : int^{l'}, \mathsf{pc} : l \sqcup l' \parallel \Lambda \vdash_\Sigma P}{\Gamma, r : int^{l'}, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{bnz}\ r, L; P}\ \mathsf{T\_CondBrnch}$$

$$\frac{\begin{array}{ll}\Gamma(r_d) = \omega^{l_d} & r_d, r_s, r_t \neq \mathsf{pc} \\ \Gamma(r_s) = int^{l_s} & l \sqcup l_s \sqcup l_t \sqsubseteq l_d \\ \Gamma(r_t) = int^{l_t} & \Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma P\end{array}}{\Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{arith}\ r_d \leftarrow r_s \odot r_t; P}\ \mathsf{T\_Arith}$$

$$\frac{\begin{array}{ll} & r_d, r_s \neq \mathsf{pc} \\ \Gamma(r_d) = \omega^{l_d} & l \sqcup l_s \sqsubseteq l_d \\ \Gamma(r_s) = int^{l_s} & \Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma P\end{array}}{\Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{arithi}\ r_d \leftarrow r_s \odot i; P}\ \mathsf{T\_Arithi}$$

$$\frac{\begin{array}{ll}\Gamma(r_s) = [\tau]^{l_s} & r_d, r_s \neq \mathsf{pc} \\ \Gamma(r_d) = \omega^{l_d} & l \sqcup l_s \sqsubseteq l_c \sqsubseteq l_d \\ \tau[c] = \omega_c^{l_c} & \Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma P\end{array}}{\Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{load}\ r_d \leftarrow r_s[c]; P}\ \mathsf{T\_Load}$$

$$\frac{\begin{array}{ll}\Gamma(r_d) = [\tau]^{l_d} & r_d, r_s \neq \mathsf{pc} \\ \Gamma(r_s) = \tau[c] = \omega^{l_s} & l \sqcup l_d \sqsubseteq l_s \\ \tau \text{ is } \mathsf{code\text{-}free} & \Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma P\end{array}}{\Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{store}\ r_d[c] \leftarrow r_s; P}\ \mathsf{T\_Store}$$

$$\frac{l \sqsubseteq \Sigma(L)(\mathsf{pc}) \quad \Gamma, \mathsf{pc} : l \parallel L \cdot \Lambda \vdash_\Sigma P}{\Gamma, \mathsf{pc} : l \parallel \Lambda \vdash_\Sigma \texttt{cpush}\ L; P}\ \mathsf{T\_Cpush} \qquad \frac{\Sigma(L) = \Gamma' \parallel \Lambda \quad \Gamma'_{/\mathsf{pc}} \subseteq \Gamma_{/\mathsf{pc}} \quad \mathsf{Ctxt}(P) \vdash_\Sigma P}{\Gamma \parallel L \cdot \Lambda \vdash_\Sigma \texttt{cjmp}\ L; P}\ \mathsf{T\_Cjmp}$$

Figure 2: Subtyping for contexts and typing rules for programs.

2.3 Typing rules

A *signature* (Σ) is a mapping assigning contexts to labels. The context $\Sigma(L)$ contains the typing assumptions for the registers in the program point pointed to by the label L. The judgment $\Gamma \parallel \Lambda \vdash_\Sigma P$ is a typing judgment for a SIF program P, with signature Σ, in a context $\Gamma \parallel \Lambda$. We say that a program P is *well-typed* if $\mathsf{Ctxt}(P) \vdash_\Sigma P$, where $\mathsf{Ctxt}(P)$ is the partial function defined as: $\mathsf{Ctxt}(L; P) = \Sigma(L)$, $\mathsf{Ctxt}(\texttt{eof}) = \{\} \parallel \epsilon$.

The typing rules for SIF programs, shown in Figure 2, are designed to prevent illegal flows of information. The directive `eof` is treated as a `halt` instruction. So, rules T_Eof and T_Halt ensure that the stack is empty.

Rule T_Label requires that the current context be compatible with the context expected at the position of the label, as defined in the signature (Σ) of the program. Jumps and conditional jumps are typed by rules T_Jmp and T_CondBrnch. In both rules the current context has to be compatible with the context expected at the destination code. In T_CondBrnch, both the code pointed to by L and the remaining program P are considered destinations of the jump included in this operation. In order to avoid implicit flows of information, the security level of the pc in the destination code should not be lower than the current security level and the security level of the register (r) that controls the branching.

In T_Arith the security level of the source registers and the pc should not exceed the security level of the target register to avoid explicit flows of information. The security level of r_d can actually be lowered to reflect its new contents, but, to avoid implicit information flows, it cannot be lowered beyond the level of the pc. Similarly for T_Arithi, T_Load and T_Store. In T_Load, an additional condition establishes that the security level of the pointer to the heap has to be lower than or equal to the security level of the word to be read.

The rule T_Cpush controls whether `cpush` L can add the code label L to the stack. Since L is going to be consumed by a `cjmp` L instruction, its security level should not be lower than the current level of the pc. The `cjmp` L instruction jumps to the junction point pointed to by label L. Furthermore, to prevent ill structured programs the rule T_Cjmp forces the code label L to be at the top of the stack, and the current context has to be

compatible with the one expected at the destination code. However, since a `cjmp` instruction allows the security level to be lowered, there are no conditions on its security level.

2.4 Type soundness of SIF

In this section we define a semantics for the untyped assembly instructions operating on a machine model, we give an interpretation for SIF types which captures the way types are implemented in memory, and finally we prove that the execution of a well-typed SIF program modifies a type-safe configuration into another type-safe configuration.

Let $\mathsf{Reg} = \{0, 1, \ldots, \mathsf{R}_{\max}\}$ be the register indices, with two dedicated registers: $R(0) = 0$, and $R(\mathsf{pc})$ is the program counter. Let $\mathsf{Loc} \subseteq \mathbf{Z}$ be the set of memory locations on our machine, Wrd be the set of machine words that can stand for integers or locations, and Code be the set of machine words which can stand for machine instructions. To simplify the presentation, we assume that Wrd is disjoint from Code; so, our model keeps code separate from data.

A *machine configuration* M is a pair (H, R) where $H : \mathsf{Loc} \rightharpoonup \mathsf{Wrd} \uplus \mathsf{Code}$ is a heap configuration, and $R : \mathsf{Reg} \to \mathsf{Wrd}$ is a register configuration.

Given a program P, a *machine assembled for* P is a machine configuration which contains a representation of the assembly program, with machine instructions stored in some designated contiguous portion of the heap. Supposing $P = p_1; \ldots; p_n$, the assembly process defines a function $\mathsf{PAdr} : 1, \ldots, n \to \mathsf{Loc}$ which gives the destination location for the code when assembling the typed instruction p_u, where $1 \leq u \leq n$. For each of the locations ℓ where P is stored, $H(\ell) \in \mathsf{Code}$. The assembly process also defines the function $\mathsf{LAdr}(L)$, which assigns to each label in P the heap location where the code pointed to by the label was assembled.

Given a machine configuration $M = (H, R)$, we define a *machine transition* $M \longrightarrow M'$, as follows: First, M' differs from M by incrementing $R(\mathsf{pc})$ according to the length of the instruction in $H(R(\mathsf{pc}))$; then, the transformation given in the table below is applied to obtain the new heap H', or register bank R'. The operations on r_0 have no effect.

`jmp` L	$R' = R[\mathsf{pc} := \mathsf{LAdr}(L)]$
`bnz` r, L	$R' = \begin{cases} R, \text{ if } R(r) = 0 \\ R[\mathsf{pc} := \mathsf{LAdr}(L)], \text{ otherwise} \end{cases}$
`arith` $r_d \leftarrow r_s \odot r_t$	$R' = R[r_d := R(r_s) \odot R(r_t)]$
`arithi` $r_d \leftarrow r_s \odot i$	$R' = R[r_d := R(r_s) \odot i]$
`load` $r_d \leftarrow r_s[c]$	$R' = R[r_d := H(R(r_s) + c)]$
`store` $r_d[c] \leftarrow r_s$	$H' = H[R(r_d) + c := R(r_s)]$

$\mathsf{Asm}(p_u)$ stands for the sequence of untyped machine instructions which is the result of assembling a typed assembly instruction p_u:

$$\mathsf{Asm}(L) = \epsilon \quad \mathsf{Asm}(\texttt{eof}) = \texttt{halt} \quad \mathsf{Asm}(\texttt{cpush}\ L) = \epsilon \quad \mathsf{Asm}(\texttt{cjmp}\ L) = \texttt{jmp}\ L \quad \mathsf{Asm}(p_u) = p_u$$

We write $M \overset{\mathsf{Asm}(p_u)}{\longrightarrow} M'$, if M executes to M' through the instructions in $\mathsf{Asm}(p_u)$, by zero or one transitions in M. The reflexive and transitive closure of this relation is defined by the following rules.

$$\frac{}{M \Longrightarrow M}\ \mathsf{Refl} \qquad \frac{M_1 \overset{\mathsf{Asm}(p_u)}{\longrightarrow} M_2}{M_1 \Longrightarrow M_2}\ \mathsf{Incl} \qquad \frac{M_1 \Longrightarrow M_2 \quad M_2 \Longrightarrow M_3}{M_1 \Longrightarrow M_3}\ \mathsf{Trans}$$

2.4.1 Imposing Types on the Model

A *heap context* ψ is a function that maps heap locations to security types. A heap context contains type information about the heap locations required to type the registers. $Dom(\psi)$ denotes the domain of the heap context ψ. The

empty context is denoted by $\{\}$. We write $\psi[\ell := \tau]$ for the heap context resulting from updating ψ with $\ell : \tau$. Two heap contexts ψ and ψ' are *compatible*, denoted $\mathsf{compat}(\psi, \psi')$, if for all $\ell \in Dom(\psi) \cap Dom(\psi')$, $\psi(\ell) = \psi'(\ell)$. The following rules assign types to heap locations:

$$\frac{H(\ell) \in \mathsf{Code}}{H; \{\ell : \mathtt{code}\} \models \ell : \mathtt{code}\ \mathsf{hloc}}\ \mathsf{T\_HLocCode} \qquad \frac{H(\ell) \in \mathsf{Wrd}}{H; \{\ell : int^l\} \models \ell : int^l\ \mathsf{hloc}}\ \mathsf{T\_HLocInt}$$

$$\frac{H(\ell) \in \mathsf{Wrd} \quad \mathsf{compat}(\psi, \{\ell : [\tau]^l\}) \quad H; \psi \models H(\ell) : \tau\ \mathsf{hloc}}{H; \psi \cup \{\ell : [\tau]^l\} \models \ell : [\tau]^l\ \mathsf{hloc}}\ \mathsf{T\_HLocPtr} \qquad \frac{\mathsf{compat}(\psi, \psi') \quad H; \psi \models \ell : \tau\ \mathsf{hloc}}{H; \psi \cup \psi' \models \ell : \tau\ \mathsf{hloc}}\ \mathsf{W\_HLoc}$$

$$\frac{m_i = \mathit{size}(\sigma_0) + \ldots + \mathit{size}(\sigma_{i-1}) \quad H; \psi \models \ell + m_i : \sigma_i\ \mathsf{hloc} \quad \text{for all } 0 \leq i \leq n}{H; \psi \models \ell : \sigma_0 \times \ldots \times \sigma_n\ \mathsf{hloc}}\ \mathsf{T\_HLocProd}$$

In order to define the notion of satisfiability of contexts by machine configurations, we need to define a satisfiability relation for registers.

$$\frac{r \neq \mathsf{pc}}{M \models_{\{\}} r : int^l\ \mathsf{reg}}\ \mathsf{T\_RegInt} \qquad \frac{H; \psi \models R(r) : \tau\ \mathsf{hloc}}{(H, R) \models_{\psi} r : [\tau]^l\ \mathsf{reg}}\ \mathsf{T\_RegPtr} \qquad \frac{(H, R) \models_{\psi} r : \sigma\ \mathsf{reg} \quad \mathsf{compat}(\psi, \psi')}{(H, R) \models_{\psi \cup \psi'} r : \sigma\ \mathsf{reg}}\ \mathsf{W\_Reg}$$

A machine configuration M *satisfies* a typing assignment Γ with a heap typing context ψ (written $M \models_{\psi} \Gamma$) if and only if for each register $r_i \in Dom(\Gamma)$, M satisfies the typing statement $M \models_{\psi_i} r_i : \Gamma(r_i)\ \mathsf{reg}$, the heap contexts ψ_i are pairwise compatible, and $\psi = \cup_{\forall i} \psi_i$.

A machine configuration $M = (H, R)$ is in *final state* if $H(R(\mathsf{pc})) = \mathtt{halt}$. We define an approximation to the execution of a typed program $P = p_1; \ldots; p_n$ by relating the execution of the code locations in the machine M with the control paths in the program by means of the relation $p_u \rightsquigarrow p_v$, which holds between pairs of instructions indexed by the set:

$$\begin{array}{c} \{(i, i+1) \mid p_i \neq \mathtt{jmp}, \mathtt{cjmp}, \text{ and } i < n\} \\ \cup \\ \{(i, j+1) \mid p_i = \mathtt{jmp}\ L, \mathtt{bnz}\ r, L, \text{ or } \mathtt{cjmp}\ L, \text{ and } p_j = L\}. \end{array}$$

$p_u \overset{*}{\rightsquigarrow} p_v$ denotes the reflexive and transitive closure of $p_u \rightsquigarrow p_v$.

2.4.2 Type Soundness

In this section we show that our type system ensures that the reduction rules preserve type safety. The soundness results imply that if the initial memory satisfies the initial typing assumptions of the program, then each memory configuration reachable from the initial memory satisfies the typing assumptions of its current instruction.

The typing assumptions of each instruction of a program can be obtained from the initial context by the type-checking process. For a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$, the derivation $\mathsf{Ctxt}(P) \vdash_{\Sigma} P$ determines a sequence of contexts $\Gamma_1 \parallel \Lambda_1, \ldots, \Gamma_n \parallel \Lambda_n$ from sub-derivations of the form $\Gamma_u \parallel \Lambda_u \vdash_{\Sigma} p_u; p_{u+1}; \ldots; p_n$.

A machine configuration is considered type-safe if it satisfies the typing assumptions of its current instruction. Given a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$ and a heap context ψ, we say $M = (H, R)$ is *type safe at* u *for* P *with* ψ if M is assembled for P; $R(\mathsf{pc}) = \mathsf{PAdr}(u)$; and $M \models_{\psi} \Gamma_u$.

We prove two meta-theoretic results Progress and Subject Reduction. Progress (Theorem 1) establishes that a non-final-state type safe machine can always progress to a new machine by executing a well-typed instruction, and Subject Reduction (Theorem 2) establishes that if a type safe machine progresses to another machine, the resulting machine is also type safe.

Theorem 1 (Progress)
Suppose a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$ and a machine configuration M type safe at u. Then there exists M' such that $M \xrightarrow{\mathsf{Asm}(p_u)} M'$, or M is in *final state*.

Theorem 2 (Subject Reduction)
Suppose $P = p_1; \ldots p_u; \ldots; p_n$ is a well-typed program and (H, R) is a machine configuration type safe at u, and $(H, R) \xrightarrow{\mathsf{Asm}(p_u)} M'$. Then there exists $p_v \in P$ such that $p_u \rightsquigarrow p_v$ and M' is type safe at v.

The proof of this theorem proceeds by case analysis on the current instruction p_u, analyzing each of the possible instructions that follow p_u, based on the definition of program transitions. See the companion technical report [13] for details.

3 Non-Interference

Given an arbitrary (but fixed) security level ζ of an *observer*, non-interference states that computed low-security values ($\sqsubseteq \zeta$) should not be affected by high-security input values ($\not\sqsubseteq \zeta$). In order to prove that a program P satisfies non-interference one must show that any two terminating executions fired from indistinguishable (from the point of view of the observer) machine configurations yield indistinguishable configurations of the same security observation level.

In order to establish what it means for machine configurations to be indistinguishable from an observer's point of view whose security level is ζ, we define a ζ-indistinguishability relation for machine configurations.

The following definitions assume a given security level ζ, two machine configurations $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$, two heap contexts ψ_1 and ψ_2, and two register contexts Γ_1 and Γ_2, such that $M_1 \models_{\psi_1} \Gamma_1$ and $M_2 \models_{\psi_2} \Gamma_2$.

Two register banks are ζ-indistinguishable if the observable registers in one bank are also observable in the other, and the contents of these registers are also ζ-indistinguishable.

Definition 3.1 (ζ-indistinguishability of register banks)
Two register banks R_1 and R_2 are ζ-indistinguishable, written $\rhd_{H_1:\psi_1, H_2:\psi_2} R_1 : \Gamma_1 \approx_\zeta R_2 : \Gamma_2$ regBank, if for all $r \in Dom(\Gamma_1) \cup Dom(\Gamma_2)$, with $r \neq \mathsf{pc}$:

$$\textsc{Labl}(\Gamma_1(r)) \sqsubseteq \zeta \text{ or } \textsc{Labl}(\Gamma_2(r)) \sqsubseteq \zeta \text{ implies } \begin{cases} r \in Dom(R_1) \cap Dom(R_2) \cap Dom(\Gamma_1) \cap Dom(\Gamma_2), \\ \Gamma_1(r) = \Gamma_2(r), \text{ and} \\ \rhd_{H_1:\psi_1, H_2:\psi_2} R_1(r) \approx_\zeta R_2(r) : \Gamma_1(r) \text{ val} \end{cases}$$

Two word values v_1 and v_2 of type ω^l are considered ζ-indistinguishable, written $\rhd_{H_1:\psi_1, H_2:\psi_2} v_1 \approx_\zeta v_2 : \omega^l$ val, if $l \sqsubseteq \zeta$ implies that both values are equal. In case of pointers to heap locations, the locations have to be also ζ-indistinguishable.

Two heap values ℓ_1 and ℓ_2 of type τ are considered ζ-indistinguishable, written $\rhd_{H_1:\psi_1, H_2:\psi_2} \ell_1 \approx_\zeta \ell_2 : \tau$ hval, if $\ell_1 \in H_1$, $\ell_2 \in H_2$, and either the type τ is code and $\ell_1 = \ell_2$, or $\tau = \sigma_1 \times \ldots \times \sigma_n$ and each pair of offset locations $\ell_1 + m_i$ and $\ell_2 + m_i$ (with m_i as in rule T_HLocProd) are ζ-indistinguishable, or τ is a word type with a security label l and $l \sqsubseteq \zeta$ implies that both values are equal.

The proof of our main result, the Non-Interference Theorem 3, requires two notions of indistinguishability of stacks (Low and High). If one execution of a program branches on a condition while the other does not, the junction points stacks may differ in each of the paths followed by the executions. If the security level of the pc is low in one execution, then it has to be low in the other execution as well, and the executions must be identical. The first three rules of Figure 3 define the relation of low-indistinguishability for stacks. In low-security executions

$$\frac{}{\rhd_\Sigma \epsilon \approx_\zeta \epsilon\ \mathsf{Low}}\ \mathsf{LowAxiom} \qquad \frac{\Sigma(L)(\mathsf{pc}) \sqsubseteq \zeta \quad \rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{Low}}{\rhd_\Sigma L \cdot \Lambda_1 \approx_\zeta L \cdot \Lambda_2\ \mathsf{Low}}\ \mathsf{LowLow}$$

$$\frac{\Sigma(L_1)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \Sigma(L_2)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{Low}}{\rhd_\Sigma L_1 \cdot \Lambda_1 \approx_\zeta L_2 \cdot \Lambda_2\ \mathsf{cstackLow}}\ \mathsf{LowHigh}$$

$$\frac{\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{Low}}{\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{High}}\ \mathsf{HighAxiom} \qquad \frac{\Sigma(L)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{High}}{\rhd_\Sigma L \cdot \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{High}}\ \mathsf{HighLeft}$$

$$\frac{\Sigma(L)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{High}}{\rhd_\Sigma \Lambda_1 \approx_\zeta L \cdot \Lambda_2\ \mathsf{High}}\ \mathsf{HighRight}$$

Figure 3: ζ-indistinguishability of junction points stacks.

the associated stacks mus be of the same size, and each code label in the stack of the first execution must be indistinguishable from that of the corresponding element in the second one.

If the security level of the pc of one of the two executions is high, then the other one must be high too. The executions are likely to be running different instructions, and thus the associated stacks may have different sizes. However, we need to ensure that both executions follow branches of the same condition. This is done by requiring that both associated stacks have a common (low-indistinguishable) sub-stack. The second three rules of Figure 3 define the relation of high-indistinguishability for stacks. Also note that, as imposed by the typing rules, the code labels added to the stack associated to high-security branches are of high-security level.

Finally, we define the relation of indistinguishability of two machine con from the point of view of an observer of level ζ.

Definition 3.2
Two machine configurations $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ are *ζ-indistinguishable*, denoted by the judgment $\rhd_P M_1 : \Gamma_1, \Lambda_1, \psi_1 \approx_\zeta M_2 : \Gamma_2, \Lambda_2, \psi_2\ \mathsf{mConfig}$, if and only if

1. $M_1 \models_{\psi_1} \Gamma_1$ and $M_2 \models_{\psi_2} \Gamma_2$,
2. M_1 and M_2 are assembled for P at the same addresses,
3. $\rhd_{H_1:\psi_1, H_2:\psi_2} R_1 : \Gamma_1 \approx_\zeta R_2 : \Gamma_2\ \mathsf{regBank}$, and
4. either
   (a) $\Gamma_1(\mathsf{pc}) = \Gamma_2(\mathsf{pc}) \sqsubseteq \zeta$ and $R_1(\mathsf{pc}) = R_2(\mathsf{pc})$ and $\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{Low}$, or
   (b) $\Gamma_1(\mathsf{pc}) \not\sqsubseteq \zeta$ and $\Gamma_2(\mathsf{pc}) \not\sqsubseteq \zeta$ and $\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2\ \mathsf{High}$.

Note that both machine configurations must be consistent with their corresponding typing assignments, and they must be executing the code resulting from assembling P.

We may now state the non-interference theorem establishing that starting from two indistinguishable machine configurations assembled for the same program P, if each execution terminates, the resulting machine configurations remain indistinguishable.

In the following theorem and lemmas, for any instruction p_i in a well-typed program $P = p_1; \ldots; p_n$, the context $\Gamma_i \parallel \Lambda_i$ is obtained from the judment $\Gamma_i \parallel \Lambda_i \quad \vdash_\Sigma \quad p_i; p_n$, which is derived by a sub-derivation of $\mathsf{Ctxt}(P) \vdash_\Sigma P$.

Theorem 3 (Non-Interference)
Let $P = p_1; \ldots; p_n$ be a well-typed program, $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations such that both are *type safe* at 1 for P with ψ and

$$\rhd_P M_1 : \Gamma_1, \epsilon, \psi \approx_\zeta M_2 : \Gamma_1, \epsilon, \psi \ \mathsf{mConfig}.$$

If $M_1 \Longrightarrow M_1'$ and $M_2 \Longrightarrow M_2'$, with M_1' and M_2' in *final state*, then

$$\rhd_P M_1' : \Gamma_v, \epsilon, \psi_1 \approx_\zeta M_2' : \Gamma_w, \epsilon, \psi_2 \ \mathsf{mConfig}.$$

The technical challenge that lies in the proof of this theorem is that the ζ-indistinguishability of configurations holds after each transition step. The proof is developed in two stages. First it is proved that two ζ-indistinguishable configurations that have a low (and identical) level for the pc can reduce in a *lock step fashion* in a manner invariant to the ζ-indistinguishability property. This is stated by the following lemma.

Lemma 1 (Low-PC step)
Let $P = p_1; \ldots; p_n$ be a well-typed program, such that p_{v_1} and p_{v_2} are in P, $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations. Suppose

1. M_1 is type safe at v_1 and M_2 is type safe at v_2, for P with ψ_1 and ψ_2, respectively,
2. $\rhd_P M_1 : \Gamma_{v_1}, \Lambda_{v_1}, \psi_1 \approx_\zeta M_2 : \Gamma_{v_2}, \Lambda_{v_2}, \psi_2 \ \mathsf{mConfig}$,
3. $\Gamma_{v_1}(\mathsf{pc}) \sqsubseteq \zeta$ and $\Gamma_{v_2}(\mathsf{pc}) \sqsubseteq \zeta$,
4. $M_1 \overset{\mathsf{Asm}(p_{v_1})}{\longrightarrow} M_1'$, and
5. there exists p_{w_1} in P such that $p_{v_1} \rightsquigarrow p_{w_1}$, and M_1' is type safe at w_1 with ψ_3.

Then, there exists a configuration M_2' such that:

(a) $M_2 \overset{\mathsf{Asm}(p_{v_2})}{\longrightarrow} M_2'$,

(b) there exists p_{w_2} in P such that $p_{v_2} \rightsquigarrow p_{w_2}$, and M_2' is type safe at w_2 with ψ_4, and

(c) $\rhd_P M_1' : \Gamma_{w_1}, \Lambda_{w_2}, \psi_3 \approx_\zeta M_2' : \Gamma_{w_2}, \Lambda_{w_2}, \psi_4 \ \mathsf{mConfig}$.

When the level of the pc is low, the programs execute the same instructions (with possibly different heap and register bank). They may be seen to be *synchronized* and each reduction step made by one is emulated with a reduction of the same instruction by the other. The resulting machines must be ζ-indistinguishable.

However, a conditional branch (bnz) may cause the execution to fork on a high value. As a consequence, both of their pc become high and we must provide proof that there are some ζ-indistinguishable machines to which they reduce. Then, the second stage of the proof consists of showing that every reduction step of one execution whose pc has a high-security level can be met with a number of reduction steps (possibly none) from the other execution such that they reach indistinguishable configurations. The High-PC Step Lemma states such result.

Lemma 2 (High-PC Step)
Let $P = p_1; \ldots; p_n$ be a well-typed program, such that p_{v_1} and p_{v_2} are in P, and $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations. Suppose

1. M_1 is type safe at v_1 and M_2 is type safe at v_2, for P with ψ_1 and ψ_2, respectively.

2. $\rhd_P M_1 : \Gamma_{v_1}, \Lambda_{v_1}, \psi_1 \approx_\zeta M_2 : \Gamma_{v_2}, \Lambda_{v_2}, \psi_2$ mConfig,

3. $\Gamma_{v_1}(\mathsf{pc}) \not\sqsubseteq \zeta$ and $\Gamma_{v_2}(\mathsf{pc}) \not\sqsubseteq \zeta$,

4. $M_1 \xrightarrow{\mathsf{Asm}(p_{v_1})} M_1'$, and

5. there exists p_{w_1} in P such that $p_{v_1} \rightsquigarrow p_{w_1}$ and M_1' is type safe at w_1 with ψ_3.

Then, either the configuration M_2 diverges or there exists a machine configuration M_2' such that

(a) $M_2 \Longrightarrow M_2'$,

(b) there exists p_{w_2} in P such that $p_{v_2} \overset{*}{\rightsquigarrow} p_{w_2}$ and M_2' is type safe at w_2 with ψ_4, and

(c) $\rhd_P M_1' : \Gamma_{w_1}, \Lambda_{w_1}, \psi_3 \approx_\zeta M_2' : \Gamma_{w_2}, \Lambda_{w_2}, \psi_4$ mConfig.

The main technical difficulty here is the proof of the case when one execution does a `cjmp` instruction that lowers the pc level. In this case, the other execution should, in a number of steps, also reduce its pc level accordingly. This is guaranteed by two facts. First, high-indistinguishable stacks share a sub-stack whose top is the label to the junction point where the pc level is reduced and both executions converge. Second, well-typed programs reach final states only with an empty stack, having visited all the labels indicated by the junction point stack.

4 Related Work

Information flow analysis has been an active research area in the past three decades [18]. Pioneering work by Bell and LaPadula [4], Feiertag et al. [9], Denning and Denning [8, 7], Neumann et al. [17], and Biba [5] set the basis of multilevel security by defining a model of information flow where subjects and objects have a security level from a lattice of security levels. Such a lattice is instrumental in representing a security policy where a subject cannot read objects of level higher than its level, and it cannot write objects at levels lower than its level.

The notion of *non-interference* was first introduced by Goguen and Meseguer [10], and there has been a significant amount of research on type systems for confidentiality for high-level languages including Volpano and Smith [20], and Banerjee and Naumann [2]. Type systems for low-level languages have been an active subject of study for several years now, including TAL [14], STAL [15], DTAL [21], Alias Types [19], and HBAL [1].

In his PhD thesis [16], Necula already suggests information flow analysis as an open research area at the assembly language level. Zdancewic and Myers [22] present a low-level, secure calculus with ordered linear continuations. An earlier version of our type system was inspired by that work. However, we discovered that in a typed assembly language it is enough to have a junction point stack instead of mimicking ordered linear continuations. Barthe et al. [3] define a JVM-like low-level language with a heap and an operand stack. The type system is parameterized by control dependence regions, and it is assumed that there exist functions that obtain such regions. In contrast, SIF allows such regions to be expressed in the language by using code labels and its well-formedness to be verified during type-checking. Crary et al. [6] define a low-level calculus for information flow analysis, however, their calculus has the structuring construct `if-then-else`, unlike SIF that uses typed pseudo-instructions that are assembled to standard machine instructions.

5 Conclusions and Future Work

We defined SIF, a typed assembly language for secure information flow analysis. Besides the standard features, such as heap and register bank, SIF introduces a stack of code labels in order to simulate at the assembly level

the block structure of high-level languages. The type system guarantees that well-typed programs assembled on type-safe machine configurations satisfy the non-interference property: for a security level ζ, if two type-safe machine configuration are ζ-indistinguishable, then the resulting machine configurations after execution are also ζ-indistinguishable.

An alternative to our approach is to have a list of the program points where the security level of the `pc` can be lowered safely. This option delegates the security analysis of where the `pc` level can be safely lowered to a previous step (that may use, for example, a function to calculate control dependence regions [12]). This delegation introduces a new trusted structure into the type system. Our type system, however, does not need to trust the well-formation of such a list. Moreover, even the signature (Σ) attached to SIF programs is untrusted in our setting, since, as we explained in section 2.3, its information about the security level of the `pc` is checked in the rules for `cpush` and `cjmp` in order to prevent illegal information flows.

We are currently developing a version of our language that includes a runtime stack, in order to define a stack-based compilation function from a high-level imperative programming language to SIF.

Acknowledgments: We are grateful to Pablo Garralda, Healfdene Goguen, David Naumann, and Alejandro Russo for enlightening discussions and comments on previous drafts. This work was partially supported by the *NSF* project *CAREER: A formally verified environment for the production of secure software* – #0093362 and the Stevens Technogenesis Fund.

References

[1] David Aspinall and Adriana B. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning, Special Issue on Proof-Carrying Code*, 31(3-4):261–302, 2003.

[2] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of Fifteenth IEEE Computer Security Foundations - CSFW*, pages 253–267, June 2002.

[3] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[4] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report Technical Report MTR 2547 v2, MITRE, November 1973.

[5] K. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[6] Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. Technical Report CMU-CS-03-164, Carnegie Mellon University, September 2003.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[8] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, May 1976.

[9] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *6th ACM Symp. Operating System Principles*, pages 57–65, November 1977.

[10] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1982.

[11] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. In *Proceedings of BYTECODE, ETAPS'05, to appear*, 2005.

[12] Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102, pages 265–285. Springer-Verlag, 2001.

[13] Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. A typed assembly language for secure information flow analysis. `http://www.cs.stevens.edu/~rmedel/hbal/publications/sifTechReport.ps`, 2005.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[15] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[16] George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998.

[17] Peter G. Neumman, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. Software development and proofs of multi-level security. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 421–428. IEEE Computer Society, October 1976.

[18] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[19] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, April 2000.

[20] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[21] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology, July 1999.

[22] S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3), 2002.

Session IV

Network Security and Denial-of-Service Attacks

Trusting the Network

(Extended Abstract)

Tom Chothia
Laboratoire d'Informatique (LIX)
École Polytechnique
91128 Palaiseau Cedex, France.

Dominic Duggan
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ, USA.

Ye Wu
Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ, USA.

Abstract

Cryptography is often used to secure the secrecy and integrity of data, but its ubiquitious use (for example on every read and write of a program variable) is prohibitive. When protecting the secrecy and integrity of data, applications may choose to reply on the underlying runtime or network, or they may seek to secure the data themselves using cryptographic techniques. However specifying when to rely on the environment, and when to explicitly enforce security, is usually specified informally without recourse to explicit policies. This article considers an approach to making explicit when the runtime or network is trusted to carry data in cleartext, and when the data must be explicitly protected. The approach is based on associating levels of trust to parts of the system where data may reside in transit, and levels of relative sensitivity to data that is transmitted. The trust policy is enforced by a type system, one that distinguishes between security policies for access control and trust policies for controlling the safe distribution of data.

1 Introduction

This article addresses the question: When should one trust the network over which data is communicated?

What is a network? We take a very generic point of view: a network is any communication medium through which messages are exchanged between entities in a computing system. Examples include:

- hosts exchanging packets through the Internet;
- processes exchanging data across a virtual private network secured using IPsec;
- processes on the same machine communicating via IPC;
- threads in a single address space exchanging messages via a message queue.

We assume that the data being exchanged has both secrecy and integrity constraints associated with it. We say that a network is *trusted* if parties using that network to communicate rely on the network itself to realize the desired secrecy and integrity guarantees. For example in type-based information flow control systems, the "network" is represented by global shared variables, and a type system ensures that secrecy constraints are enforced for all participants. On the other hand, in distributed programming over the Internet, the network is not trusted; so cryptographic techniques are used to secure secrecy and integrity properties (encryption and signing, respectively). There is still some trust: cryptographic operations may not be performed until data is about to be output from the network card, since the operating system and its buffers are assumed to be trusted. Virtual private networks use cryptographic techniques to build a virtual trusted network that entities can use to communicate with assurance that the requisite properties are satisfied.

In most of these examples, the desired secrecy and integrity properties, and the level of trust in the network, are expressed relatively informally. There is then little hope of relating the two to make sure that there is enough trust in the network to enforce these properties, and if not to use cryptographic techniques to achieve those properties. In type-based information flow systems and distributed programming over the Internet, it appears clear where to place trust (everywhere and nowhere, respectively). But there are gradations between these extremes: typed threads may communicate through an operating system component that is outside the type system, or across an untrusted network; hosts may communicate across a trusted subnet behind a firewall.

In this article we provide formalizations of both secrecy and integrity properties, and of the trust placed in parts of the network, and we relate them in such a way that the properties are guaranteed to be achieved (provided, that is, one has not placed trust in an unreliable part of the network).

Various models have been proposed for specifying secrecy and integrity properties of information in the type system [15, 16, 8]. These systems associate sets of policies (called *labels*) with variables, along with their types. This article works with a much simpler model, where a label simply specifies a set of principals (actually four sets of principals, each for a different purpose). Nevertheless the system presented here could be applied to the aforesaid information flow control systems. Our system is simplified because we do consider notions such as declassification. The novelty of this work is in a model of trust for networks, and how that is related to secrecy and integrity constraints on data exchanged over a network.

Work has previously considered type-based APIs to cryptographic operations [1, 11, 8] that relate secrecy and integrity policies, as defined in program types, with the use of cryptographic operations to dynamically enforce those policies where necessary (when transmitting data over the Internet, for example). However absent from these models is when these operations *must* be performed. Requiring them to be always performed is in practice ridiculous: every write of a variable would require encryption and signing, and every read of a variable would require decryption and authentication. However sometimes these operations must be performed, e.g., when non-trivial policies must be enforced over a raw TCP/IP channel. Implicit in these extremes is where the trust is placed in the network.

We formalize this notion of trust using a notion of *zones*. A zone is an abstraction of any notion of network location, be it a process address space, a host, or a physical or virtual network. Entities specify that certain zones are trusted for certain communications. If data is transmitted across a zone that is trusted, then policies can be left to be enforced by the type system. For example, a general may send a message to soldiers in the field commending them or informing them of promotion; type-based techniques can guarantee that the general is the originator of the message. If data is transmitted across an untrusted zone, then the policies must be cryptographically enforced. For example, the general may place less trust in the network when sending a command to field commanders for a coordinated attack. On the other hand, if the general is in conference with field commanders within a secure conference room, it may be sufficient to rely on lightweight type-based security for such local communications.

We use the term *trust policy* to distinguish it from *security policy*. We use the latter to refer to access control policies to enforce secrecy and integrity constraints. We use the former to refer to levels of trust that are placed in parts of the network to respect such security policies. There should be no confusion with the term "trust management," although our approach could obviously be enriched with notions of delegation.

Trust and security policies are largely orthogonal. Security policies specify fine-grained read and write permissions for data. Trust policies specify the relative trustworthiness of parts of the network for enforcing those security policies, without application intervention using cryptography. Both forms of policies are specified using a type system. The type system ensures that security policies are respected by all well-typed processes. It also ensures that data with non-trivial security policies is not transmitted in cleartext over parts of the network that are not trusted to only contain well-typed processes.

1. A principal may be included in the trust secrecy policy for a piece of data but not the security secrecy policy. This simply means that processes for that principal may execute in zones where the principal is not allowed

to access the data; the principal is "trusted" to respect the security policy (as far as that piece of data is concerned).

2. A principal may be included in the security secrecy policy for a piece of data but not the trust secrecy policy. The principal is allowed to access the data, but cannot execute at any zones where the data is transmitted in cleartext. This is admittedly strange, but harmless, and demonstrates the orthogonality of trust and secrecy policies. There are extensions of our system, involving declassification, where this scenario may be useful.

We give an informal introduction to zones and trust policies in the next section. In Sect. 3 we provide a formalization of the type system. Finally Sect. 4 considers related work, while Sect. 5 provides conclusions.

2 Informal Motivation

The type system we introduce incorporates notions of *principals*, *labels* and *policies*. Every program variable has both a type and a label. A label $L = (\pi_1, \pi_2)$ is a pair of *policies*, one for secrecy and the other for integrity. A policy $\pi = (\{\overline{P_1}\}, \{\overline{P_2}\})$ is a pair of sets of principals:

1. The first set specifies the *security policy*, the set of principals that are allowed to access that variable (reading the variable in the case of a secrecy policy, and writing into that variable in the case of an integrity policy).

2. The second set specifies the *trust policy*, the set of principals that are expected to respect the security policy. The security policies are only enforced statically at sites that are accessible to principals included in the corresponding trust policies. At all other sites, cryptographic techniques must be used to enforce the security policies dynamically.

The security policy is enforced by the type system. At load time (e.g., via bytecode verification in a Java Virtual Machine), programs are checked by type-checking to ensure that the security policies are respected. We are assuming a model where code is signed and the loader checks code signatures before running them. Thus a process does not attempt to read a variable unless the principal for which it executes (the principal that signed its code base) is in the set of principals specified by the read security policy. Any program for which this check fails is rejected by the loader.

In a perfect world, all running processes are typed and security policies are always respected. Zones allow relative levels of trust to be placed on parts of the network. If data is transmitted through a zone that is not trusted, then cryptographic techniques must be used to protect it. Therefore, orthogonal to security policies, data also has trust policies associated with it.

Zones are an abstractions for network and network trust. A zone is abstractly a location that contains processes and messages. A process may only receive messages that are in its own zone. A process can send a message locally (within its own zone). A process may also send a message to another zone (routing). Routers are modelled just as normal user-space programs that accept messages and forward them to other zones. For simplicity we assume a fully-connected graph of network connections, though it would be trivial to enforce more restricted connectedness.

A zone has a *level*, reflecting the level of trust that is placed in it. A level is denoted simply as a set of principals. The only processes that can send and receive messages in a zone are those executing for the principals in the corresponding zone level.

Zone levels effectively enforce which principals are allowed access to a zone, for sending or for receiving. An insider attack consists of running "untyped" code in a zone. For an outside attacker to access a zone, it must compromise a principal allowed in that zone and then mount an insider attack. Network trust policies can then be defined based on (a) which principals can execute processes in a zone and (b) how susceptible those principals' processes are to insider or outsider attacks. There is implicit in this model some notion of authenticating the right of code to execute in a zone. We simplify our model by assuming that the only code that runs in a zone is

that which is allowed by the zone's level. The actual details of authentication can be assumed to be part of the trusted computing base (TCB). An interesting direction for further work is to see how this authentication could be reflected from application programs into the TCB.

2.1 Examples

Fig. 1 gives an example of processes executing for principals, and implicitly the zones for those principals. We assume a zone level CJIS (Criminal Justice Information Division) partitioned into two sublevels: NCIC (National Crime Information Center) and NICS (National Instant Criminal Background Check System). NCIC allows processes for two principals, A and B, to execute there, while NICS allows processes for principal C. Another zone level, PAPD (Port Authority Police Department), allows processes for principals X and Y to execute there. Processes for principals B and X are required to be able to share data, therefore we establish an intermediate zone level CPD (Central Police Desk) for just those two principals. CPD describes all sites on which only processes for those principals can execute. This could be implemented as a trusted path, a physical firewall, or a distributed firewall or VPN established using cryptographic techniques. The framework introduced here does not depend on the details of how access to zones is established. Processes for B may execute in the same zones as processes for A and C (at levels NCIC and NICS). Similarly processes for X may execute in the same zone as processes for Y (at level PAPD). Messages exchanged between processes for B and processes for X may be passed in cleartext in zones of level CPD.

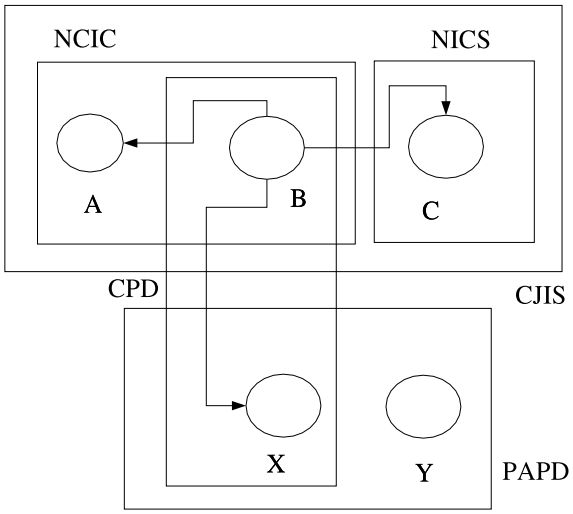


Figure 1: Principals and Zone Levels

Now whether messages exchanged between B and X may be passed in cleartext in zones of level NCIC and PAPD, for example, depends on the relative sensitivity of the data and the relative level of trust placed in such zone levels. We can label data with the level of its relative sensitivity, and then compare this with the level of a zone through which the data is transmitted.

We assume that policies are specified as sets of principals. This is clearly a simplifying assumption that could be removed by introducing some notion of levels or roles in the security system, and relying on authorization certificates to allow people to assume particular roles. Then policies could be specified in terms of such roles or levels. However for simplicity in the article we stay with sets of principals. For the example above we can define the following:

```
prinset NCIC = {A,B};
prinset NICS = {C};
prinset CJIS = NCIC ∪ NICS;
prinset PAPD = {X,Y};
prinset CPD = CJIS ∩ PAPD;
prinset World;
```

We can treat `World` as a special marker for the set of all principals. Now using these definitions of sets of principals, we can specify the following policies:

```
policy TopSecret = ( CPD, CPD );
policy FBI_Secret = ( CPD, NCIC ∪ {X} );
policy Internal = ( NCIC ∪ PAPD, CJIS ∪ PAPD );
policy Public = ( World, World );
```

For example FBI_Secret defines information that is only intended for CPD personnel to see, but we trust members of the FBI to carry this information without violating confidentiality.

We focus in this example on secrecy. We define the following labels where we elide the integrity policies:

```
label TopSecretL = ( TopSecret, ...  );
label FBI_SecretL = ( FBI_Secret, ...  );
label InternalL = ( Internal, ...  );
```

A packet where the payload has label TopSecretL and the address has label InternalL expresses that the payload is top secret, but we allow the address to be visible within the organization, so it may be viewed for example by couriers that carry information back and forth. This example requires the payload to be encrypted outside CPD. A more lenient policy retains the notion that the payload is top secret, but enforces this based on the trustworthiness of the principals in the environment rather than relying on encryption. If we label the payload as FBI_SecretL and the address as InternalL, then we allow top secret information to be transmitted by processes for FBI (CJIS) personnel who may not have authority to read the information that they are conveying, however we do not trust unauthorized PAPD processes to respect the access restrictions.

In these examples the message-passing communication system checks the access control specifications on the data being transmitted with the mapping from zones names to sets of principals, to detect violations of the policy specified in the types. Since the former mapping will change after programs have been compiled, some amount of dynamic checking is unavoidable[1]. Nevertheless in the remainder of the article we rely on static checking using a type system. This at least formulates the properties that any kind of type checking is seeking to achieve, and leaves open the possibility of some hybrid regimen of static and dynamic policy checking.

To model attacks by adversaries who do not respect the restrictions of the type system, we assume a special principal Eve. There are many possibilities for the kinds of attacks we can allow this adversary to mount. At one extreme processes executing for Eve may be completely untyped. However it becomes difficult (not impossible, but somewhat complex) to say useful things about the ability of the system to withstand such attacks. In any case, untyped attacks based on bypassing the type system may often be caught in this message-passing context during the unmarshalling stage of communication. We assume a weaker attack model therefore: Eve respects the type system but can ignore the labels on the types of data and she can ignore the access control checks, both for secrecy and integrity. For example, Eve may mount an attack on Alice by sending her a channel with the payload label

$$(\{\mathsf{Alice},\mathsf{Bob}\},\ldots),(\{\mathsf{Bob}\},\ldots))$$

The security parts of the policies assure Alice that this is a private channel between her and Bob, created by Bob. We consider in the next section how Alice can defend herself against this attack using trust policies. For now we note that the type system ensures that if Eve does obtain access to a piece of data, then Eve must have been one of the principals allowed in the original trust policy. In other words, if an attacker obtains access to a data by subverting the type system, the original owner of the data must have mistakenly placed its trust in an unreliable principal.

3 Type System: Formal Description

The syntax of types is provided in Fig. 2. In this language without cryptographic operations, there are two kinds of data: message channels (for message-passing communication) and tuples (for data structures). Data have both types and *labels*, where a label is a pair of *policies*: the *secrecy policy* and the *integrity policy*. Each of these policies is in turn separated into security and trust components, which in this system are simply sets of principals.

[1]This dynamic checking would consist of checking that the trust restrictions on data being routed to a different zone are satisfied by the set of principals that are allowed to execute at that zone.

$T \in \text{Type}$	::=	$\mathsf{Chan}(LT)$	Channel
	$\mid$	$\langle LT_1, \ldots, LT_k \rangle$	Tuple
$\pi \in \text{Policy}$	::=	$(\{\overline{P_1}\}, \{\overline{P_2}\}$	
$L \in \text{Label}$	::=	(π_1, π_2)	
$LT \in \text{Labelled type}$	::=	T^L	

Figure 2: Syntax of Sensitivity Types

$v \in \text{Value}$	::=	w, x, y, z	Variable
	$\mid$	a, b, c, n	Channel name
	$\mid$	$\langle V_1, \ldots, V_k \rangle$	Tuple
	$\mid$	V^L	Annotated value
$R \in \text{Process}$	::=	stop	Stopped process
	$\mid$	$\mathsf{let}_L\ x = \langle v_1, \ldots, v_k \rangle; R$	Create tuple
	$\mid$	$\mathsf{let}\ \langle x_1, \ldots, x_k \rangle = v; R$	Decompose tuple
	$\mid$	$\mathsf{let}_L\ x = v; R$	Annotate value
	$\mid$	$\mathsf{receive}\ c?x;\ R$	Message receive
	$\mid$	$\mathsf{send}\ v!v$	Message send
	$\mid$	$\mathsf{send}\ \ell.v!v$	Message route
	$\mid$	$\mathsf{new}(a : LT); R$	New channel
	$\mid$	$(R_1 \mid R_2)$	Parallel composition
$N \in \text{Network}$	::=	empty	Empty network
	$\mid$	$(P, \ell)[R]$	Located process
	$\mid$	$\mathsf{new}(a : LT)N$	Channel binding
	$\mid$	$(N_1 \mid N_2)$	Wire

Figure 3: Syntax of Values, Processes and Networks

For a trust policy, the set of principals in the secrecy component denotes those principals who are *trusted to honor secrecy restrictions for the data*, while the set of principals in the integrity component denotes those who are *trusted to honor integrity restrictions for the data*. We highlight this explanation because *trust policies do not themselves specify secrecy and integrity restrictions*. These restrictions should be expressed using the security policy. It is straightforward to extend our simple security policies with the other kinds of policy languages.

The syntax of values, processes and networks is given in Fig. 3. Values include variables, channel names and tuples. Processes include parallel composition (for forking new processes), new channel creation, operations for building and taking apart tuples, and basic message-passing operations (blocking receive and non-blocking send). This is a two-level syntax: every process executes under the authority of a principal and at a particular zone (network location). So the network is the parallel composition of a collection of located processes, each of the

$VE \vdash v : T^L$	Well-formed value
$VE \vdash (P, \ell)[R]$	Well-formed process
$VE \vdash N$	Well-formed network

Figure 4: Judgements for Type System

form $(P, \ell)[R]$ where P is the principal, ℓ the zone and R the process. The purpose of this two-level syntax is twofold:

- to enforce security policies based on the authority of the principal under which a process executes; and
- to support a *routing operation* that allows a process at one zone to route a message to another zone.

We provide a type system using judgements of the form given in Fig. 4. The type system for processes uses judgements of the form $VE \vdash (P, \ell)[R]$ to check that the process R is well-formed, under the assumption that it will be evaluated (executed) under the authority of the principal P, at the zone ℓ, with free names bound in the environment VE. An environment is a sequence of pairs, binding variables or names to types:

$$VE \in \text{Value Env} \quad ::= \quad \{\} \mid \{(x : LT)\} \mid \{(a : LT)\} \mid VE_1 \cup VE_2$$

The types of values, processes and networks are provided in the full version of the paper [9].

As noted, we denote attacker processes as those executing for the special principal Eve. An example was provided in Sect. 2. Eve is still subject to checks on where in the network her processes can execute. Alice can prevent the forgery attack described in Sect. 2 by only accepting a private channel sent in a zone that does not allow Eve to execute processes (in the integrity part of the trust policy). But then Alice and Bob can only communicate in the same trusted space. The more interesting cases then are where cryptography is used to secure communication over untrusted spaces, and where Eve may still attempt to mount attacks based on interception and forgery of cryptographic keys.

In the extension of the system with cryptographic operations, there are now types for encrypted and signed data, $\mathcal{E}\{T\}$ and $\mathcal{S}\{T\}$ respectively. There are also types for public and private keys, for encryption and signing. Each of these key types is indexed by a policy. This reflects the intuition that a key fundamentally is used to enforce a policy across address spaces. If decryption of ciphertext succeeds, then the secrecy policy associated with the key type is re-established for the resulting cleartext. Similarly if authentication of signed ciphertext succeeds, then the integrity policy associated with the key type is re-established for the resulting cleartext.

Theorem 1 (Type Preservation) *Suppose* $VE \vdash N_1$ *and* $N_1 \xrightarrow{TE;VE} N_2$, *then* $VE \vdash N_2$.

A network N is "stuck" if no evaluation rule is applicable to it.

Theorem 2 (Progress) *If a network N is stuck, then the remaining processes are of the form:*

1. *A receive operation with no matching send message to synchronize with.*
2. *A decrypt operation where the decryption key is not the inverse of the encryption key for the ciphertext.*
3. *An authentication operation where the authentication key is not the inverse of the signing key for the ciphertext.*

In particular this justifies omitting run-time access checks in the semantics. Note that the checks for processes' permissions to execute in zones is determined statically in the type system, as mentioned earlier. This static restriction could be relaxed with the addition of zone variables to the language.

We still need to say something about the extent to which processes of Eve may subvert the security of the system. We focus on names (channel names and cryptographic keys) as the values whose secrecy is paramount. Say that a name is *leaked* if there is a free occurence as a subterm of a process for Eve, where that occurence does not occur as part of a piece of ciphertext.

Theorem 3 *Given VE and N, with* $(c : T^L) \in VE$ *and c is leaked in N. Then* $\mathsf{Eve} \in SLEV(L)$.

Intuitively the secrecy trust policy for the data reflects all possible principals in all possible zones where the data may be transmitted in cleartext. If the data is encrypted, then the type rule for encryption keys requires that the decryption key have at least as strict a secrecy trust level. If the data or key is transmitted directly from a process of Eve in the current zone, even with annotation, Eve must be included in the set of principals that can execute in the zone where the key was created. If the data or key is transmitted via an intermediary (trusted) process, then that process or some ancestor in a chain of intermediaries exchanged the data or key in a zone with Eve; and since none of the intermediaries have access to the annotation operation, the trust secrecy policy for the data or key must include Eve as a principal.

4 Related Work

The motivation for this work has been the need for proper programming abstractions for applications that must manage the task of securing their own communication. Much of the work on abstractions for Internet programming has focused on security, for example, providing abstractions of secure channels [4, 3], controlling key distribution [7], reasoning about security protocols [1, 5], tracking untrustworthy hosts in the system [14, 19], etc.

The work of Riely and Hennessy [14, 19] in particular has some relationship to this work. They provide a type system that reflects the relative level of trust in hosts in the network. They are motivated by ensuring that mobile agents do not migrate to untrusted hosts. An "untrusted" host in our system amounts to a zone that includes a process executing for the attacker Eve. This extra level of indirection is more than cosmetic, since it reflects our concern with enforcing access control policies through a combination of static and dynamic techniques, with trust policies used to determine when dynamic techniques (cryptography) must be used.

Abadi [1] considers a type system for ensuring that secrecy is preserved in security protocols. For securing communication over untrusted networks, he includes a "universal" type `Un` inhabited by encrypted values. His type system prevents "secrets" from being leaked to untrusted parties, but allows encrypted values to be divulged. In an analogous way, encrypted values in our type system provide a way to temporarily subvert the access controls in the type system, with the secrecy properties enforced by labels reasserted when the ciphertext is decrypted/authenticated. Gordon and Jeffrey [12, 13] have developed a type-based approach to verifying authentication protocols.

Abadi and Blanchet [2, 6] have worked on analyzing security protocols, showing how it is possible to guarantee secrecy properties and then generalizing this to guarantee integrity. Their system uses a type of "secret," and a type system that ensures that secret items are never put on channels that are shared with untrusted parties. They can translate types in their system into logic programs that can then be used to check protocols for correctness. The emphasis of this work is somewhat different, since Bruno and Blancet work in a more "black and white" environment where there are trusted parties and untrusted parties. In contrast our interest is in a more refined type system where we allow certain parties to access certain data, and where different levels of trust are placed in different parts of the network.

Other work on security in programming languages has focused on ensuring safety properties of untrusted code [17] and preventing unwanted security flows in programs [10, 15, 21, 18]. Sabelfeld and Myers [20] provide an excellent overview of work in language-based information-flow security. Our security concerns have largely been with access control, but the work can be extended with ideas from the decentralized label model of JIF [16, 8].

Our work is clearly related to that of J/Split [9, 22]. That system partitions a sequential JIF program into a distributed system, where portions of the program run on hosts that are trusted for the principals for whom the code runs. There is then a tight relationship between the access restrictions on data (specified using the richer form of access restrictions allowed by JIF) and the hosts where data may be stored in cleartext. For two mutually distrustful principals engaged in a distributed game, for example, a trusted third party (the board) is responsible for communicating data from one party to the other. Network security (cryptographic operations) is implicitly part of the TCB.

Our model can be viewed as attempting to expose some of the TCB machinery of this approach to the application, with the eventual goal of modeling the J/Split runtime as application programs in our approach. Thus although hosts can be viewed as analogous to zones (both are simply abstractions of network locations, hardly a new concept), we decouple the security and trust policies because each means different things.

5 Conclusions

The ultimate goal of this research program is to provide a programming environment where secrecy and integrity requirements are specified explicitly in the type system, where these requirements are related to the relative trustworthiness of parts of the network. Security policy specifies what must be protected; trust policy specifies how it must be protected. Finally a type-based API to cryptographic operations relates the use of these operations to the requirements that they are intended to satisfy.

The antithesis of such an environment is one where all security requirements are enforced inside the runtime. This leads to a bloated trusted computing base (TCB) and flies in the face of the well-known end-to-end argument in system design. An interesting possible avenue for the application of our approach is in Web services authentication, where end-to-end security considerations predominate [5].

We have deliberately chosen a very simple language for presenting our approach. There are numerous avenues to pursue with this work. Clearly it can be combined with the full extent of KDLM [8], which provides a somewhat richer policy language, borrowing ideas from JIF [16] and adding declassification certificates. A more interesting direction to consider is allowing zone variables, so that processes can build routing tables and perform dynamic routing decisions, while continuing to perform static checking as much as possible. Zone types based on zone levels, perhaps building on the work of Riely and Hennessy [14, 19], appear to be a promising direction in this regard.

References

[1] Martin Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Science*, pages 611–638, 1997.

[2] Martin Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 33–44, 2002.

[3] Martin Abadi, Cedric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, 1999.

[4] Martin Abadi, Cedric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2000.

[5] Karthikeyan Bhargavan, Cedric Fournet, and Andrew D. Gordon. A semantics for web services authentication. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2004.

[6] Bruno Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, pages 242–259, 2002.

[7] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *Concurrency Theory (CONCUR)*. Springer-Verlag, 2000.

[8] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Computer Security Foundations Workshop*, 2003.

[9] Tom Chothia, Dominic Duggan, and Ye Wu. Trusting the network. `http://guinness.cs.stevens.edu/~dduggan/Public/Papers/zones.pdf`, 2005.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 1977.

[11] Dominic Duggan. Cryptographic types. In *Computer Security Foundations Workshop*, Nova Scotia, Canada, 2002. IEEE Press.

[12] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2001.

[13] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[14] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[15] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, 1998.

[16] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[17] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, 1996.

[18] Francois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of ACM International Conference on Functional Programming*, 2000.

[19] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[20] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2002.

[21] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, 1997.

[22] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.

Formal Modeling and Analysis of DoS Using Probabilistic Rewrite Theories*

Gul Agha, Michael Greenwald, Carl A. Gunter, Sanjeev Khanna
Jose Meseguer, Koushik Sen, and Prasannaa Thati†

Abstract

Existing models for analyzing the integrity and confidentiality of protocols need to be extended to enable the analysis of availability. Prior work on such extensions shows promising applications to the development of new DoS countermeasures. Ideally, it should be possible to apply these countermeasures systematically in a way that preserves desirable properties already established. This paper investigates a step toward achieving this ideal by describing a way to expand term rewriting theories to include probabilistic aspects that can be used to show the effectiveness of DoS countermeasures. In particular, we consider the shared channel model, in which adversaries and valid participants share communication bandwidth according to a probabilistic interleaving model, and a countermeasure known as selective verification applied to the handshake steps of the TCP reliable transport protocol. These concepts are formulated in a probabilistic extension of the Maude term rewriting system, called PMAUDE. Furthermore, we formally verified the desired properties of the countermeasures through automatic statistical model-checking techniques.

1 Introduction

There are well-understood models on which to base the analysis of integrity and confidentiality. The most common approaches are algebraic techniques [6] based on idealized cryptographic primitives and complexity-theoretic techniques [5] based on assumptions about complexity. There has also been progress on unified perspectives that enable using the simpler algebraic techniques to prove properties like those ensured by the more complete cryptographic techniques. However, neither of these approaches or their unifications are designed to approach the problem of availability threats in the protocols they analyze. For example, suppose a protocol begins with a sender sending a short message to a receiver, where the receiver's first step is to verify a public key signature on the message. A protocol like this is generally considered to be problematic because an adversarial sender can send many packets with bad signatures at little cost to himself while the receiver will need to work hard to (fail to) verify these signatures. Algebraic and complexity-theoretic analysis techniques ensure only that the recipient will not be fooled by the bad packets and will not leak information as a result of receiving them. However, they do not show that the receiver will be available to a valid sender in the presence of one or more attackers.

In [7] we began an effort to explore a formal model for the analysis of DoS based on a simple probabilistic model called the "shared channel" model. This effort showed that the shared channel model could be used to prove properties of DoS countermeasures for authenticated broadcast that could be verified in experiments. We have subsequently conducted a number of experiments to explore the application of such countermeasures to other classes of protocols. The aim of this paper is to explore the prospects for using the shared channel model as a

*This work was supported in part by ONR Contract N00014-02-1-0715.

†Addresses of the authors: K. Sen, G. Agha, C. A. Gunter, J. Meseguer, University of Illinois at Urbana-Champaign; Michael Greenwald, Lucent Bell Labs; Sanjeev Khanna, University of Pennsylvania; Prasannaa Thati, Carnegie-Mellon University,

foundation for extending term rewriting models of network protocols to cover DoS aspects of the protocols and their modification with counter-measures. Our particular study is to investigate the use of a probabilistic extension of the Maude rewrite system called PMAUDE and its application to understanding the effectiveness of a DoS countermeasure known as "selective sequential verification" [7]. This technique was explored for authenticated broadcast in [7] but in the current paper we consider its application to handshake steps of the TCP reliable transport protocol.

At a high level, our ultimate aim is to demonstrate techniques for showing how a network protocol can be systematically "hardened" against DoS using probabilistic techniques while preserving the underlying correctness properties the protocol was previously meant to satisfy. Specifically, given a protocol P and a set of properties T, we would like to expand T to a theory T^* that is able to express availability properties and show that a transformation P^* of P meets the constraints in T^* without needing to re-prove the properties T that P satisfied in the restricted language. The shared channel model provides a mathematical framework for this extension.

In this paper, we develop a key element of this program: a formal language in which to express the properties T^* and show that availability implications hold for P^*. We attempt to validate this effort by showing its effectiveness on a selective verification for TCP. In particular, we show how we can specify TCP/IP 3-way handshake protocol in PMAUDE algebraically. First, we take a previously specified formal non-deterministic model of the protocol. We then replace all non-determinism by probabilities. The resulting model with quantified non-determinism (or probabilities) is then analyzed for quantitative properties such as availability. The analysis is done by combining Monte-Carlo simulation of the model with statistical reasoning. In this way, we leverage the existing modelling and reasoning techniques to quantified reasoning without interfering with the underlying non-quantified properties of the model.

The rest of the paper is organized as follows. In Section 2, we give the preliminaries of DoS theory followed by its application to TCP/IP 3-way handshaking protocol in Section 3. Then we briefly describe PMAUDE in Section 4. In Section 5, we describe and discuss the algebraic probabilistic specification of DoS hardened TCP/IP protocol in PMAUDE. We describe the results of our analysis of some desired properties written in the query language for the specification of TCP/IP protocol in Section 6.

2 DoS Theory

On the face of it, the conventional techniques for establishing confidentiality and integrity are inappropriate for analyzing DoS, since they rely on very strong models of the adversary's control of the network. In particular, they assume that the adversary is able to delete packets from the network at will. An adversary with this ability has an assured availability attack. Typical analysis techniques therefore adapt this assumption in one of two ways. A first form of availability analysis is to focus on the relationship between the sender and the attacker and ask whether the attacker/sender is being forced to expend at least as much effort as the valid receiver. In our example, this is an extremely disproportionate level of effort, since forming a bad signature is much easier than checking that it is bad. Thus the protocol is vulnerable to the imposition of a disproportionate effort by the receiver. This is a meaningful analysis, but it does not answer the question of whether a valid sender will experience the desired availability. A second form of availability analysis is to ask whether the receiver can handle a specified load. For instance, a stock PC can check about 8000 RSA signatures each second, and it can receive about 9000 packets (1500 bytes per packet) each second over a 100Mbps link. Thus a receiver is unable to check all of the signatures it receives over such a channel. A protocol of the kind we have envisioned is therefore deemed to be vulnerable to a *signature flood* attack based on cycle exhaustion. By contrast, a stock PC can check the hashes on 77,000 packets each second, so a receiver that authenticates with hashes can service all of its bandwidth using a fraction of its capacity. This sort of analysis leads one to conclude that a protocol based on public key signatures is vulnerable to DoS while one based on hashes is not.

These techniques are sound but overly conservative, because they do not explicitly account for the significance of *valid* packets that reach the receiver. Newer techniques for analyzing DoS have emerged in the last year that provide a fresh perspective by accounting for this issue. In essence, these new models are both more realistic for the Internet and suggest new ideas for countermeasures. We refer to one basic version of this new approach as the *shared channel model*. The shared channel model is a four-tuple consisting of the minimum bandwidth W_0 of the sender, the maximum bandwidth W_1 of the sender (where $W_0 \leq W_1$), the bandwidth α of the adversary, and the loss rate p of the sender where $0 \leq p < 1$. The ratio $R = \alpha/W_1$ is the *attack factor* of the model. When $R = 1$, this is a *proportionate* attack and, when $R > 1$, it is a *disproportionate* attack. As in the algebraic model, the adversary is assumed to be able to replay packets seen from valid parties and flood the target with anything he can form from these. But in the shared channel model he is not able to delete specific packets from the network. In effect, he is able to interleave packets among the valid ones at a specified maximum rate. This interleaving may contribute to the loss rate p of the sender, but the rate of loss is assumed to be bounded by p and randomly applied to the packets of the sender.

The key insight that underlies the techniques in this paper arises from recognizing the *asymmetry* the attacker aims to exploit; his willingness to spend his entire bandwidth on an operation that entails high cost for the receiver also offers opportunities to burden the attacker in disproportionate ways relative to the valid sender. This can be seen in a simple strategy we call *selective verification*. The idea is to cause the receiver to treat the signature packets she receives as arriving in an *artificially* lossy channel. The sender compensates by sending extra copies of his signature packets. If the recipient checks the signature packets she receives with a given probability, then the number of copies and the probability of verification can be varied to match the load that the recipient is able to check. For example, suppose a sender sends a 10Mbps stream to a receiver, but this is mixed with a 10Mbps stream of DoS packets devoted entirely to bad signatures. To relieve the recipient of the need to check all of these bad signatures, the receiver can check signatures with a probability of 25%, and, if the sender sends about 20 copies of each signature packet, the receiver will find a valid packet with a probability of more than 99% even if the network drops 40% of the sender's packets. This technique is inexpensive, scales to severe DoS attacks, and is adaptable to many different network characteristics.

3 SYN Floods as DoS for TCP/IP

TCP is an extremely common reliable bi-directional stream protocol that uses a three-way handshake to establish connections. Glossing over many details, a sender initiates a connection by sending a packet with the SYN flag set and an initial sequence number. The receiver responds by acknowledging the SYN flag, and sending back a SYN with its *own* sequence number. When the original sender acknowledges the receiver's SYN (this ACK is the 3rd packet in a 3-way handshake), then the connection is ESTABLISHED.

Each established connection requires a TCB (Transmission Control Block) at each end of the connection. The TCB occupies a few hundred bytes of identification and control information, statistics, as well as a much larger allocation of packet buffers for received data and (re)transmission queues. In most operating system kernels, both packet buffer space and the number of available TCBs are fixed at boot time, and they constitute a limited resource. This opens a significant vulnerability to adversaries who aim to overwhelm this limit by flooding a server with SYN packets; this is typically called a *SYN flood attack*. This threat is mitigated in many systems by storing connection information in a SYN cache (a lighter-weight data structure, recording only identity information and sequence numbers for the connection) until the connection becomes ESTABLISHED, at which point the (more expensive) full TCB is allocated. Normally, a legitimate connection occupies a slot in the SYN cache for only one round trip time (RTT). If no ACK for the SYN+ACK arrives, then the server eventually removes the entry from the SYN cache, but only after a much longer timeout interval, t_A.

SYN flooding constitutes an easy denial of service attack because SYN cache entries are relatively scarce, while the bandwidth needed to send a single SYN packet is relatively cheap. The attacker gains further leverage from

the disparity between the one RTT slot occupancy (often on the order of a millisecond or less) for a legitimate client, compared with a fraudulent SYN packet that typically holds a syn-cache slot for a value of t_A ranging from 30-120 seconds.

A SYN attack is simple to model; attackers merely send SYN packets at a cumulative rate which we denote by r_A. We can compute the effectiveness of the DoS attack by the probability of success of a client's attempt to connect, and from that compute the number of legitimate connections per second that the server can support under a given attack rate r_A. If the server offers no defense, and if the order in which incoming SYNs are processed at the server is adversarially chosen, then it is clear that an attack rate r_A of $O(B/t_A)$ suffices to completely take over a syn-cache of size B. To see this, observe that in every second, B/t_A of the attacker's slots in the SYN cache expire, and B/t_A new ones arrive to take their places. Even in a more realistic model where the incoming SYN requests are assumed to be ordered in accordance with a random permutation, it is easy to show that an attack rate of $O(B/t_A)$ suffices.

It is clear from this analysis (as well as from abundant empirical evidence) that even a moderate rate of DoS attack can totally disable a server. For a server with a SYN cache of size $B = 10,000$ and a timeout interval of 75 seconds a moderate attack rate of 200 to 300 SYNS per second is enough to almost completely overwhelm the server! (An energetic attacker can generate SYN packets 1000 times as quickly as this on a commodity 100Mbps Fast Ethernet link.)

Selective verification can improve this performance significantly[1]. Let B denote the number of slots in the SYN cache. Suppose we want to ensure that the attacker never blocks more than a fraction f of the table, for $0 < f < 1$. We ask the server to process each incoming SYN with probability p where p satisfies $pt_A r_A \leq fB$, then we ensure that at least a $(1 - f)$-fraction of the SYN cache is available to legitimate users. We effectively inflate the bandwidth cost of mounting an attack rate of r_A to be r_A/p. Considering once again an attacker on 100 Mbps channel ($300,000$ SYNs/sec), if we set $p = 10^{-3}/6$, we ensure that the attacker cannot occupy more than half the table at any point in time. The attacker can still deny service, but is now required to invest as much in bandwidth resources as the collective investment of the clients that it is attacking.

If we increase the cache size by a factor of 30, we can get an identical guarantee with $p = .005$. The overhead on a valid client to establish a connection then is only 200 SYN packets, roughly 8KB, for each request. These overheads are not insignificant but they allow us to provide unconditional guarantees on availability of resources for valid clients. If we downloaded the PS version of this paper (500KB), the blowup increases the transfer size by 2%. Moreover, these overheads should be contrasted with the naive alternative: the cache size would have to be increased to 6×10^7 to get the same guarantee.

4 Probabilistic Rewrite Theories

Rewriting logic is an expressive semantic framework to specify a wide range of concurrent systems [11]. In practice, however, some systems may be probabilistic in nature, either because of their environment, or by involving probabilistic algorithms by design, or both. This raises the question of whether such systems can also be formally specified by means of rewrite rules in some suitable probabilistic extension of rewriting logic. This would provide a general formal specification framework for probabilistic systems and could support different forms of symbolic simulation and formal analysis. In particular, DoS-resistant communication protocols such as the DoS-hardened TCP/IP protocol discussed in Section 3 could be formally specified and analyzed this way.

The notion of a probabilistic rewrite theory provides an example of such a semantic framework. Usually, the rewrite rules specifying a non-probabilistic system are of the form

[1]Techniques such as SYN cookies are also effective against SYN flooding; however they do not preserve the underlying behavior of TCP.

$$t \Rightarrow t' \quad \texttt{if}\ C$$

where the variables appearing in t' are typically a subset of those appearing in t, and where C is a condition. The intended meaning of such a rule is that if a fragment of the system's state is a substitution instance of the pattern t, say with substitution θ, and the condition $\theta(C)$ holds, then our system can perform a local transition in that state fragment changing it to a new local state $\theta(t')$. Instead, in the case of a probabilistic system, we will be using rewrite rules of the form,

$$t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}, \overrightarrow{y})\ \texttt{if}\ C(\overrightarrow{x})\ \texttt{with probability}\ \overrightarrow{y} := \pi_r(\overrightarrow{x})$$

where the first thing to observe is that the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ appearing in t. Therefore, such a rule is *non-deterministic*; that is, the fact that we have a matching substitution θ such that $\theta(C)$ holds, does not uniquely determine the next state fragment: there can be many different choices for the next state depending on how we instantiate the extra variables $\overrightarrow{y}$. In fact, we can denote the different such next states by expressions of the form $t'(\theta(\overrightarrow{x}), \rho(\overrightarrow{y}))$, where θ is fixed as the given matching substitution, but ρ ranges along all the possible substitutions for the new variables $\overrightarrow{y}$. The probabilistic nature of the rule is expressed by the notation `with probability` $\overrightarrow{y} := \pi_r(\overrightarrow{x})$, where $\pi_r(\overrightarrow{x})$ is a probability distribution *which depends on the matching substitution* θ, and we then choose the values for $\overrightarrow{y}$, that is the substitution ρ, probabilistically according to the distribution $\pi_r(\theta(\overrightarrow{x}))$.

We can illustrate these ideas with a very simple example, namely a digital battery-operated clock that measures time in seconds. The state of the clock is represented by a term `clock(t,c)`, where `t` is the current time in seconds, and `c` is a rational number indicating the amount of charge in the battery. The clock ticks according to the following probabilistic rewrite rule:

$$\texttt{clock(t,c)} \Rightarrow \texttt{if B then clock(t + 1,c-}\tfrac{\texttt{c}}{1000}\texttt{) else broken(t,c -}\ \tfrac{\texttt{c}}{1000}\texttt{) fi}$$
$$\texttt{with probability B := BERNOULLI(}\tfrac{\texttt{c}}{1000}\texttt{)}\ .$$

Note that the rule's righthand side has a new boolean variable `B`. If all goes well (`B = true`), then the clock increments its time by one second and the charge is slightly decreased; but if `B = false`, then the clock will go into a broken state `broken(t,c -` $\frac{\texttt{c}}{1000}$`)`. Here the boolean variable `B` is distributed according to the Bernoulli distribution with mean $\frac{\texttt{c}}{1000}$. Thus, the value of `B` *probabilistically depends on the amount of charge* left in the battery: the lesser the charge level, the greater the chance that the clock will break; that is, we have different probability distributions for different matching substitutions θ of the rule's variables (in particular, of the variable `c`).

Of course, in this example the variable `B` is a discrete binary variable; but we could easily modify this example to involve continuous variables. For example, we could have assumed that `t` was a real number, and we could have specified that the time is advanced to a new time `t + t'`, with `t'` a new real-valued variable chosen according to an exponential distribution. In general, the set of new variables $\overrightarrow{y}$ could contain both discrete and continuous variables, ranging over different data types. In particular, both discrete and continuous time Markov chains can be easily modeled, as well as a wide range of discrete or continuous probabilistic systems, which may also involve nondeterministic aspects [9]. Furthermore, the PMAUDE extension of the Maude rewriting logic language allows us to symbolically simulate probabilistic rewrite theories [10, 3], and we can formally analyze their properties according to the methods described in [3]. Due to space constraints, we do not give the mathematical definition of probabilistic rewrite theories. Readers are referred to [10, 9] for such details.

In general, a probabilistic rewrite theory $\mathcal{R}$ *involves both probabilities and non-determinism*. The non-determinism is due to the fact that, in general, *different rules, possibly with different subterm positions and substitutions* could be applied to rewrite a given state u: the choice of what rule to apply, and where, and with which substitution is *non-deterministic*. It is only when such a choice has been made that probabilities come into the picture, namely for choosing the substitution ρ for the new variables $\overrightarrow{y}$. In particular, for the kind of statistical

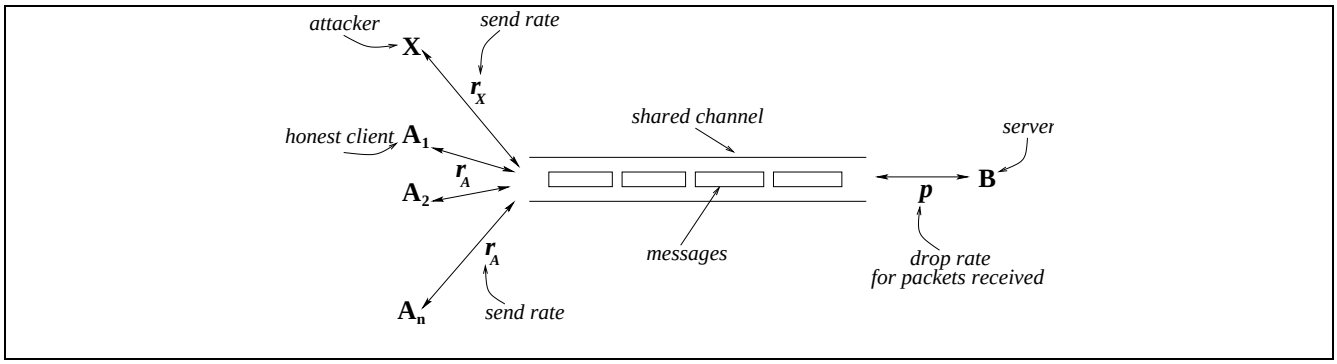


Figure 1: An instance of the TCP's 3-way handshake protocol.

model checking discussed in [13, 14] that will be used to formally analyze our DoS-resistant TCP/IP protocol, we need to assume that *all non-determinism has been eliminated* from our specification; that is, that at most one single rule, position, and substitution are possible to rewrite any given state.

What this amounts to, in the specification of a concurrent system such as a network protocol, is the *quantification of all non-determinism due to concurrency using probabilities*. This is natural for simulation purposes and can be accomplished by requiring the probabilistic rewrite theory to satisfy some simple requirements described in [3].

We will consider rewrite theories specifying concurrent actor-like objects [2] and communication by asynchronous message passing; this is particularly appropriate for communication protocols. In rewriting logic, such systems (see [12] for a detailed exposition) have a distributed state that can be represented as a *multiset* of objects and messages, where we can assume that objects have a general record-like representation of the form: $\langle$`name:` $o \mid a_1 : v_1, \ldots a_1 : v_1\rangle$, where o is the object's name and the $a_i : v_i$ its corresponding attribute-value pairs in a given state. It is also easy to model in this way *real-time concurrent object systems*: one very simple way to model them is to include a global clock as a special object in the multiset of objects and messages. Rewrite rules in such a system will involve an object, a message, and the global time and will consume the message, change the object's state, and send messages to other objects. To deal with message delays and their probabilistic treatment, we can represent messages as *scheduled objects* that are inactive until their associated delay has elapsed.

5 Probabilistic Rewrite Specification of DoS resistant TCP 3-way Handshaking

We now present an executable specification of TCP's 3-way handshake protocol in probabilistic rewriting logic. We consider a protocol instance composed of N honest clients $C_1, \ldots, C_N$ trying to establish a TCP connection with the server S, and a single attacker A that launches a SYN-flood attack on S (see Figure 1). The clients C_i transmit SYN requests to S at the rate r_C, while the attacker A floods spurious SYN requests at the rate r_A. These rates are assumed to be parameters of an exponential distribution from which the time for sending the next packet is sampled. The server S drops each packet it receives, independently, with probability p. We assume that each message across the network is subject to a constant transmission delay d. Of course, these assumptions about the various distributions can be easily changed in the implementation that follows.

Each client C_i is modeled as an object with four attributes as follows.

```
<name: C(i) | isn:N, repcnt:s(CNT), sendto:SN, connected:false>
```

The attribute `isn` specifies the sequence number that is to be used for the TCP connection, `sendto` specifies the name of server S, `repcnt` specifies the number of times the SYN request is to be (re)transmitted in order to account for random dropping of packets at S, and `connected` specifies if the connection has been successfully established as yet. The attacker is modeled as an object with a single attribute as follows.

```
<name: AN | sendto: SN >
```

The server S is modeled as an object with two attributes.

```
<name: SN | isn: M , synlist: SC >
```

The attribute `isn` specifies the sequence number that S uses for the next connection request it receives, while `synlist` is the SYN cache that S maintains for the pending connection requests.

Following is the probabilistic rewrite rule that models the client C_i sending a SYN request.

```
<name:C(i) | isn:N, repcnt:s(CNT), sendto:SN, connected:false> (C(i)← poll) T
          ⇒ <name: C(i) | isn:N, repcnt:CNT, sendto:SN, connected:false>
            [ d + T , (SN← SYN(C(i),N)) ] [ t + T , (C(i)← poll) ]   T
                                  with probability t := EXPONENTIAL(r_C)  .
```

We use special poll messages to control the rate at which C_i retransmits the SYN requests. Specifically, C_i repeatedly sends itself a poll message, and each time it receives a poll message it sends out a SYN request to S. The poll messages are subject to a random delay t that is sampled from the exponential distribution with parameter r_C. Specifically, the message is scheduled at time $t + T$, where T is the current global time. The net effect of this is that C_i sends SYN requests to S at rate r_C. Perhaps it is important to point out that the poll messages are not regular messages that are transmitted across the network; they have been introduced only for modeling purposes. Further, note that the approach of simply freezing C_i by scheduling it at time $T + t$ does not work since that would also prevent C_i from receiving any SYN+ACK messages that it may receive from S meanwhile. Finally, note that the replication count is decremented by one after the transmission of SYN message, and the message itself is scheduled with a delay d.

The rule for SYN flooding by the attacker is very similar, except that it uses randomly generated sequence numbers.

```
<name: AN |  sendto: SN > (AN ←  poll) T ⇒ <name: AN | sendto: SN >  T
          [ d + T , (SN← SYN(AN,random(counter))) ] [ t + T , (AN← poll) ]
                                  with probability t := EXPONENTIAL(r_A)  .
```

The following rule models the processing of SYN requests by the server S.

```
<name: SN | isn: M , synlist: SC > (SN← SYN(ANY,N)) T
⇒ if(drop? or size(SC) > SYN-CACHE-SIZE) then <name: SN | isn: M,synlist: SC > T
  else <name: SN | isn:s(M), synlist:add(SC,entry(ANY,M))>
       [d+T,(ANY← SYN+ACK(SN,N,M))] [TIMEOUT+T,(SN← tmout(entry(ANY,M)))] T fi
                                  with probability drop? := BERNOULLI(p)  .
```

The random dropping of incoming messages is modeled by sampling from the Bernoulli distribution with the appropriate parameter p. An incoming request can also be dropped if the SYN cache is full. If the cache is not full, for each request that is not dropped, the server S makes an entry for the request in the cache, and sends out a SYN+ACK message to the source of the request. A cache entry is of the form `entry(N,M)` where `N` is the name of the source which has requested a connection, and `M` is the sequence number for the connection. Timing out of entries in the cache is modeled by locally sending a message to self that is scheduled after an interval of time equal to the timeout period. Here is the rule for removing timed out entries.

```
<name: SN|isn: N,synlist: [s(SZ),(L1 entry(ANY,M) L2)]> (SN ←
tmout(entry(ANY,M)))
       ⇒   <name: SN | isn: N , synlist: [ SZ , (L1 L2) ] >  .
```

The first argument in the value of the `synlist` attribute above is the number of entries in the list, while the second argument is the actual list of entries. The rule for processing the SYN+ACK message at the clients is as follows.

```
 <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:false> (C(i)← SYN+ACK(SN,N,M)) T
 ⇒                <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> [d+T,(SN←
ACK(C(i),M))] T  .
```

The rule is self-explanatory; the only significant point to be noted is that the attribute `connected` is set to true after processing the SYN+ACK message. Since the clients replicate their requests to account for random dropping of packets at the server, it is possible for them to receive a SYN+ACK message for a connection that has already been established. Such SYN+ACK messages are simply ignored as follows.

```
 <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> (C(i)← SYN+ACK(SN,N,M))
 ⇒  <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> .
```

In contrast to the honest clients, the attacker ignores all the SYN+ACK messages that it receives from the server S.

```
  <name: AN|sendto:SN > (AN← SYN+ACK(SN,N,M)) ⇒  <name: AN|sendto:SN >  .
```

Finally, the initial configuration of the system is

```
< name: AN | ... >   [ t1 , < name: C(1) | ... > ] [ t2 , < name: C(2) | ... > ] ...
    [tn , < name: C(N) | ... > ] < name: SN | ... >
```

where $t_1, \ldots, t_n$ are all distinct and positive. Note that, since all the clients are scheduled at different times, it follows [3] that the system does not contain any un-quantified non-determinism, which is essential for statistical analysis to be possible.

6 Analysis

We have successfully used the statistical model-checking tool VESTA [13, 14] to verify various desired properties of the probabilistic model in Section 5. In the following, we first describe the tool VESTA and its integration with PMAUDE. We then elaborate on the verification of one important property of the 3-way handshake protocol presented in the previous section.

The integration of PMAUDE and VESTA is described in detail in [3]. In the integrated tool, we assume that VESTA is provided with a set of sample execution paths generated through the discrete-event simulation of a PMAUDE specification with no non-determinism. We assume that an execution path that appears in our sample is a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots$, where s_0 is the unique initial state of the system, s_i is the state of the system after the i^{th} computation step (rewrite), and t_i is the difference of global time between the states s_{i+1} and s_i. We also assume that there is a labelling function L that assigns to each state s_i a set of atomic propositions that hold in that state; the set of atomic propositions are all those that appear in the property of interest (see below). Thus, $L : S \rightarrow 2^{AP}$, where AP is a set of relevant atomic propositions and S is the set of system states. In PMAUDE, this labelling function is defined as an operator that maps terms representing states to sets of atomic propositions.

In VESTA, we assume that the properties are expressed in a sublogic of Continuous Stochastic Logic – CSL (without stationary state operators). CSL was introduced in [1] as a logic to express probabilistic properties. The syntax and the semantics of the logic and the statistical model-checking algorithm for CSL are described in [13, 14]. In our experiments, we model checked the following property expressed in CSL for different values of the attacker rate r_A.

$$\mathbf{P}_{\leq 0.01}(\Diamond(\textit{successful\_attack}()))$$

where *successful_attack*() is true in a state if the SYN cache of S is full, i.e., the attacker has succeeded in launching the SYN flood attack. The property states that the probability that eventually the attacker A successfully fills up the SYN cache of S is less than 0.01.

The results of model-checking are shown in the following table for two cases: in the absence of DoS counter-measure and in the presence of DoS counter-measure with the parameter p set to 0.9. In all experiments,

we used scaled down parameters so that our experiments could be completed in a reasonable amount of time. Specifically, we used a SYN cache size of 10,000, cache timeout of 10 seconds, and 100 clients. The experiments were carried out on 1.8 GHz Xeon Server with 2 GB RAM and running Mandrake Linux 9.2.

Model-checking $\mathbf{P}_{\leq 0.01}(\Diamond(\textit{successful\_attack}()))$		X's attack rate (SYNs per second)								
		1	5	64	100	200	400	800	1000	1200
$p = 0.0$ (No counter-measure)	result	F	F	F	T	T	T	T	T	T
	time (10^2 sec)	47	87	280	605	183	183	182	182	181
$p = 0.9$ (With counter-measure)	result	F	F	F	F	F	F	F	T	T
	time (10^2 sec)	68	75	217	328	896	3102	11727	2281	1781

The results show that in the presence of DoS counter-measure with $p = 0.9$, S can sustain an attack from A with attack rate 10 times larger than that in the case of no counter-measure. Therefore, the results validate our hypothesis that *selective verification* can be used as an effective counter-measure for DoS attacks.

To gain more insight into the probabilistic model, we realized that model-checking is not sufficient. Specifically, we found the *true* (T) and *false* (F) answers given by the model-checker is not sufficient to understand the various quantitative aspects of the probabilistic model. For example, we wanted to know the expected number of clients that get connected in the presence of SYN flood attack. Therefore, in addition to model-checking, we used a query language called *Quantitative Temporal Expressions* (or QUATEX in short). The language is mainly motivated by probabilistic computation tree logic (PCTL) [8] and EAGLE [4]. In QUATEX, some example queries that can be encoded are as follows:

1. What is the expected number of clients that successfully connect to S out of 100 clients?

2. What is the probability that a client connected to S within 10 seconds after it initiated the connection request?

A detailed discussion of the QUATEX is beyond the scope of this paper. However, we provide a brief introduction of QUATEX in the Appendix.

We evaluated the following QUATEX expression with different values of the attacker rate r_A.

$$\texttt{CountConnected}() = \underline{\text{if}}\ \textit{completed}()\ \underline{\text{then}}\ \textit{count}()\ \underline{\text{else}}\ \bigcirc\ (\texttt{CountConnected}())\ \underline{\text{fi}};$$
$$\underline{\text{eval}}\ \mathbf{E[}\texttt{CountConnected}()\mathbf{]}$$

In this expression, $\textit{completed}()$ is true in a state if all the clients C_i have either sent all of their SYN packets or have managed to connect with S. The expression $\textit{count}()$ in a state returns the number of clients that have successfully connected to S. The expression queries the expected number of clients that eventually connect with S in the presence of DoS attack by the attacker A.

The results of evaluating the above expression for different values of attacker rate r_A are plotted in Figure 2. The results show that most of the clients get connected as long as the attacker does not manage to fill up the SYN cache buffer. However, as soon as the attacker's SYN rate becomes high enough to fill the SYN cache buffer, none of the clients gets connected. The plot also illustrates that with *selective verification* the server can withstand an order of magnitude higher SYN flood rates than without.

7 Conclusions

We have presented a general framework for verification of DoS properties of communication protocols. We are able to express and prove key properties, but performance limitations of the automated system in our current formulation require us to use scaled down version of parameters that arise in practice. Addressing these efficiency limitations and verifying the properties for general systems remain future work objectives.

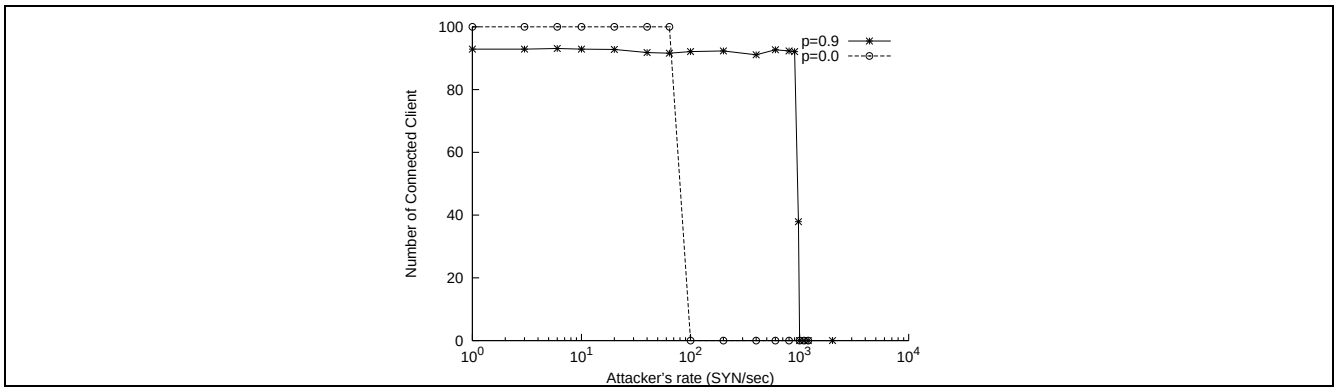


Figure 2: Expected number of clients out of 100 clients that get connected with the server under DoS attack

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102, pages 269–276.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, January 2004.

[5] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. *Lecture Notes in Computer Science*, 1462, 1998.

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

[7] C. A. Gunter, S. Khanna, K. Tan, and S. Venkatesh. Dos protection for reliably authenticated broadcast. In *Network and Distributed System Security (NDSS '04)*. Internet Society, 2004.

[8] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[9] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, Univ.of Illinois at Urbana-Champaign, 2003.

[10] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In *Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, volume 2884 of *LNCS*, pages 32–46, 2003.

[11] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[12] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[13] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215, 2004.

[14] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th Conference on Computer Aided Verification (CAV'05)*, LNCS (To Appear). Springer, 2005.

A QUATEX

We introduce the notation that describes the syntax and the semantics of QUATEX followed by a few motivating examples. Then we describe the language formally, along with an example query that we have used to investigate if the DoS free 3-way TCP/IP handshaking protocol model meets our requirements. The results of our query on various parameters are given in Section 6.

We assume that an execution path is an infinite sequence

$$\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$$

where s_0 is the unique initial state of the system, typically a term of sort `Config` representing the initial global state, s_i is the state of the system after the i^{th} computation step. If the k^{th} state of this sequence cannot be rewritten any further (i.e. is absorbing), then $s_i = s_k$ for all $i \geq k$.

We denote the i^{th} state in an execution path π by $\pi[i] = s_i$. Also, denote the suffix of a path π starting at the i^{th} state by $\pi^{(i)} = s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow \cdots$. We let $Path(s)$ be the set of execution paths starting at state s. Note that, because the samples are generated through discrete-events simulation of a PMAUDE model with no non-determinism, $Path(s)$ is a measurable set and has an associated probability measure. This is essential to compute the expected value of a path expression from a given state.

A.1 QUATEX through Examples

The language QUATEX, which is designed to query various quantitative aspects of a probabilistic model, allows us to write temporal query expressions like temporal formulas in a temporal logic. It supports a framework for parameterized recursive temporal operator definitions using a few primitive non-temporal operators and a temporal operator ($\bigcirc$). For example, suppose we want to know *"the probability that along a random path from a given state, the client $A(0)$ gets connected with B within 100 time units."* This can be written as the following query

```
IfConnectedInTime(t) = if t > time() then 0 else if connected() then 1
                                    else ◯ (IfConnectedInTime(t)) fi fi;
eval E[IfConnectedInTime(time() + 100)];
```

The first four lines of the query define the operator $\texttt{IfConnectedInTime}(t)$, which returns 1, if along an execution path $A(0)$ gets connected to B within time t and returns 0 otherwise. The state function *time()* returns the global time associated with the state; the state function *connected*() returns true, if in the state, $A(0)$ gets connected with B and returns false otherwise. Then the state query at the fifth line returns the expected number of times $A(0)$ gets connected to B within 100 time units along a random path from a given state. This number lies in $[0, 1]$ since along a random path either $A(0)$ gets connected to B within 100 time units or $A(0)$ does not get connected to B within 100 time units. In fact, this expected value is equal to the probability that along a random path from the given state, the client $A(0)$ gets connected with B within 100 time units.

A further rich query that is interesting to our probabilistic model is as follows

```
NumConnectedInTime(t, count) = if t > time() then count
            else if anyConnected() then ◯ (NumConnectedInTime(t, 1 + count))
                                    else ◯ (NumConnectedInTime(t, count)) fi fi;
eval E[NumConnectedInTime(time() + 100, 0)]
```

In this query, the state function $anyConnected()$ returns true if any client $A(i)$ gets connected to B in the state. We assume that in a given execution path, at any state, at most one client gets connected to B, which is true with our probabilistic model. We use a simpler variant of this query in our experiments.

A.2 Syntax of QUATEX

The syntax of QUATEX is given in Fig. 3. A query in QUATEX consists of a set of definitions D followed by a query of the expected value of a path expression $PExp$. In QUATEX, we distinguish between two kinds

$$
\begin{array}{rcl@{\qquad}rcl}
Q & ::= & D \ \underline{\text{eval}}\ \mathbf{E[}PExp\mathbf{]}; & SExp & ::= & c \mid f \mid F(SExp_1, \ldots, SExp_k) \mid x_i \\
D & ::= & \text{set of } Defn & PExp & ::= & SExp \mid \bigcirc N(SExp_1, \ldots, SExp_n) \\
Defn & ::= & N(x_1, \ldots, x_m) = PExp; & & & \mid \underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}
\end{array}
$$

Figure 3: Syntax of QUATEX

$$
\begin{array}{rl}
(s)[\![c]\!]_D & = c \\
(s)[\![f]\!]_D & = f(s) \\
(s)[\![F(SExp_1, \ldots, SExp_k)]\!]_D & = F((s)[\![SExp_1]\!]_D, \ldots, (s)[\![SExp_k]\!]_D) \\
(s)[\![\mathbf{E[}PExp\mathbf{]}]\!]_D & = \mathbf{E[}(\pi)[\![PExp]\!]_D \mid \pi \in Paths(s)\mathbf{]} \\
(\pi)[\![\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}]\!]_D & = \text{ if } (\pi[0])[\![SExp]\!]_D = \text{true then } (\pi)[\![PExp_1]\!]_D \text{ else } (\pi)[\![PExp_2]\!]_D \\
(\pi)[\![\bigcirc N(SExp_1, \ldots, SExp_m)]\!]_D & = \\
 & (\pi^{(1)})[\![B[x_1 \mapsto (\pi[0])[\![SExp_1]\!]_D, \ldots, x_m \mapsto (\pi[0])[\![SExp_m]\!]_D]]\!]_D \\
\text{where} & N(x_1, \ldots, x_m) = B \in D
\end{array}
$$

Figure 4: Semantics of QUATEX

of expressions, namely, *state expressions* (denoted by $SExp$) and *path expressions* (denoted by $PExp$); a path expression is interpreted over an execution path and a state expression is interpreted over a state. A definition $Defn \in D$ consists of a definition of a *temporal operator*. A temporal operator definition consists of a name N and a set of formal parameters on the left-hand side, and a path expression on the right-hand side. The formal parameters denote the *freeze formal parameters*. When using a temporal operator in a path expression, the formal parameters are replaced by state expressions. A state expression can be a constant c, a function f that maps a state to a concrete value, a k-ary function mapping k state expressions to a state expression, or a formal parameter. A path expression can be a state expression, a next operator followed by an application of a temporal operator already defined in D, or a conditional expression $\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}$. We assume that expressions are properly typed. Typically, these types would be `integer`, `real`, `boolean` etc. The condition $SExp$ in the expression $\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}$ must have the type `boolean`. The temporal expression $PExp$ in the expression $\mathbf{E[}PExp\mathbf{]}$ must be of type `real`. We also assume that expressions of type `integer` can be coerced to the `real` type.

A.3 Semantics of QUATEX

Next, we give the semantics of a subset of query expressions that can be written in QUATEX. In this subclass, we put the restriction that the value of a path expression $PExp$ that appears in any expression $\mathbf{E[}PExp\mathbf{]}$ can be determined from a finite prefix of an execution path. We call such temporal expressions *bounded* path expressions. The semantics is given in Fig. 4. $(\pi)[\![PExp]\!]_D$ is the value of the path expression $PExp$ over the path π. Similarly, $(s)[\![SExp]\!]_D$ is the value of the state expression $SExp$ in the state s. Note that if the value of a bounded path expression can be computed from a finite prefix π_{fin} of an execution path π, then the evaluations of the path expression over all execution paths having the common prefix π_{fin} are the same. Since a finite prefix of a path defines a basic cylinder set (i.e. a set containing all paths having the common prefix) having an associated probability measure, we can compute the expected value of a bounded path expression over a random path from a given state. In our analysis tool, we estimate the expected value through simulation instead of calculating it exactly based on the underlying probability distributions of the model. The exact procedure can be found at `http://osl.cs.uiuc.edu/~ksen/vesta2/`.

Session V

Invited Talk II

Constructive Authorization Logics

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
http://www.cs.cmu.edu/~fp/

Authorization logics are traditionally used to specify access control policies. More recently, in the proof-carrying authorization architecture, they have also been employed directly to enforce policies via explicit checking of proofs expressed in the logic. Authorization logics provide a great deal of flexibility, but this may make it difficult for principals to understand the consequences of their policy decisions and for users to obtain the necessary proof objects.

In ongoing work we investigate a new constructive foundation for authorization logics which makes it easier to construct them modularly and to reason about them mechanically. At the core is the separation of judgments from propositions and a lax modality indexed by principals. We have formally verified some properties of the core logic itself, such as cut-elimination, and we are now interested in methods for establishing properties of policies expressed in the logic, such as independence and non-interference.

In this talk we explain the underlying design philosophy and the indexed lax logic at the heart of our approach. We also give a brief survey of the technical results obtained so far.

Joint work with Deepak Garg and Kevin Watkins.

Session VI

Security Protocols and Decidability Issues

A Constraint-Based Algorithm for Contract-Signing Protocols

Detlef Kähler and Ralf Küsters
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel, 24098 Kiel, Germany
{kaehler,kuesters}@ti.informatik.uni-kiel.de

Abstract

Research on the automatic analysis of cryptographic protocols has so far mainly concentrated on reachability properties, such as secrecy and authentication. Only recently it was shown that certain game-theoretic security properties, such as balance for contract-signing protocols, are decidable in a Dolev-Yao style model with a bounded number of sessions but unbounded message size. However, this result does not provide a practical algorithm as it merely bounds the size of attacks. In this paper, we prove that game-theoretic security properties can be decided based on standard constraint solving procedures. In the past, these procedures have successfully been employed in implementations and tools for reachability properties. Our results thus pave the way for extending these tools and implementations to deal with game-theoretic security properties.

1 Introduction

One of the central results in the area of automatic analysis of cryptographic protocols is that the security of cryptographic protocols is decidable when analyzed w.r.t. a finite number of sessions, without a bound on the message size, and in presence of the so-called Dolev-Yao intruder (see, e.g., [14, 1]). Based on this result, many fully automatic tools (see, e.g., [2, 7, 13]) have been developed and successfully been applied to find flaws in published protocols, where most of these tools employ so-called *constraint solving procedures* (see, e.g., [13, 7, 4]). However, the mentioned decidability result and tools are restricted to security properties such as authentication and secrecy which are reachability properties of the transition system associated with a given protocol. In contrast, crucial properties required of contract-signing and related protocols (see, e.g., [9, 3]), for instance abuse-freeness [9] and balance [5], are game-theoretic properties of the structure of the transition system associated with a protocol. Balance, for instance, requires that in no stage of a protocol run, the intruder or a dishonest party has both a strategy to abort the run and a strategy to successfully complete the run and thus obtain a valid contract.

Only recently [11], the central decidability result mentioned above was extended to such game-theoretic security properties, including, for instance, balance. However, similar to the result by Rusinowitch and Turuani [14] for reachability properties, the decision algorithm presented in [11] is merely based on the fact that the size of attacks can be bounded, and hence, all potential attacks up to a certain size have to be enumerated and checked. Clearly, just as in the case of reachability properties, this is completely impractical. For reachability properties, one has therefore developed the mentioned constraint solving procedures to obtain practical decision algorithms.

The main contribution of the present work is a *constraint-based* decision algorithm for the game-theoretic security properties of the kind considered in [11]. The main feature of our algorithm is that it can be built on top of *standard constraint solving procedures* (see, e.g., [13, 7, 4] and references therein). As mentioned, such procedures have successfully been employed for reachability properties in the past and proved to be a good basis for practical implementations. Hence, our algorithm paves the way for extending existing implementations and tools for reachability properties to deal with game-theoretic security properties.

In a nutshell, our constraint-based algorithm works as follows: Given a protocol along with the considered game-theoretic security property, first the algorithm guesses what we call a symbolic branching structure. This

structure represents a potential attack on the protocol and corresponds to the interleavings, which are, however, linear structures, guessed for reachability properties. In the second step of the algorithm, the symbolic branching structure is turned into a so-called constraint system. This step requires some care due to the branching issue and write-protected channels considered in our model (also called secure channels here), i.e., channels that are not under the control of the intruder. Then, a standard constraint solving procedure (see above) is used to compute a finite sound and complete set of so-called simple constraint systems. A simple constraint system in such a set represents a (possibly infinite) set of solutions of the original constraint system and the sound and complete set of these simple constraint systems represents the set of *all* solutions of the original constraint system. Finally, it is checked whether (at least) one of the computed simple constraint systems in the sound and complete set passes certain additional tests.

There are some crucial differences of our constraint-based algorithm to algorithms for reachability properties: First, as mentioned, instead of symbolic branching structures, for reachability properties only interleavings, i.e., linear structures, need to be guessed. Turning these interleavings into constraint systems is immediate due to the absence of the branching issue and the absence of secure channels. Second, and more importantly, for reachability properties it suffices if the constraint solving procedure only returns one simple constraint system, rather than a sound and complete set. Third, the final step of our constraint-based algorithm—performing additional tests on the simple constraint system—is not required for reachability properties.

We emphasize that even though for reachability properties it suffices if the constraint solving procedure returns only one simple constraint system, standard constraint solving procedures are typically capable of computing sound and complete sets of simple constraint systems. Any such procedure can be used by our constraint-based algorithm as a black-box for solving constraint systems. This makes it possible to extend existing implementations and tools for reachability properties to deal with game-theoretic properties since the core of the algorithms—solving constraint systems—remains the same, provided that the considered cryptographic primitives can be dealt with by the constraint solving procedure (see Section 4).

The protocol and intruder model that we use is basically the one proposed in [11], which in turn is the "bounded session" version of a model proposed in [5]. We slightly modify the model of [11]—without changing its expressivity and accuracy—in order to simplify our constraint-based algorithm (see Section 2 and 3). For instance, while in [11] intruder strategies are positional, it turns out that the constraint-based algorithm is considerably simpler for intruder strategies which may depend on the history of the protocol run. However, it is not hard to show that both notions of strategies are equivalent in our setting, and hence, we can w.l.o.g. choose the notion of strategy that fits best for our purpose.

Further related work. Contract-signing and related protocols have been analyzed both manually [5], based on a relatively detailed model (as mentioned, our model is a "bounded session" version of this model), and using finite-state model checking (see, e.g., [15, 12]), based on a coarser finite-state model. Drielsma and Mödersheim [8] were the first to apply an automatic tool based on constraint solving to the contract-signing protocol by Asokan, Shoup, and Waidner [3]. Their analysis is, however, restricted to reachability properties since game-theoretic properties cannot be handled by their tool. The results shown in the present work pave the way for extending such tools in order to be able to analyze game-theoretic properties.

Structure of this paper. In Section 2, we recall the protocol and intruder model and in Section 3 the intruder strategies and the game-theoretic properties first introduced in [11], and point out the mentioned differences. Section 4 provides the necessary background on constraint solving. In Section 5, we present our constraint-based decision algorithm along with an example and state our main result—soundness, completeness, and termination of the algorithm.

Full definitions and proofs can be found in our technical report [10].

2 The Protocol and Intruder Model

The protocol and intruder model that we use basically coincides with the model first introduced in [11], which in turn is the "bounded session" version of the model proposed in [5]. We only slightly modify the model in [11] in that we impose a restriction on principals which is necessary for principals to perform feasible computations.

In our model, a protocol is a finite set of principals and every principal is a finite tree, which represents all possible behaviors of the principal, including all subprotocols a principal can carry out. Each edge of such a tree is labeled by a rewrite rule, which describes the receive-send action that is performed when the principal takes this edge in a run of the protocol.

When a principal carries out a protocol, it traverses its tree, starting at the root. In every node, the principal takes its current input, chooses one of the edges leaving the node, matches the current input with the left-hand side of the rule the edge is labeled with, sends out the message which is determined by the right-hand side of the rule, and moves to the node the chosen edge leads to. While in the standard Dolev-Yao model (see, e.g., [14]) inputs to principals are always provided by the intruder, in our model inputs can also come from the secure channel, which the intruder does not control, i.e., the intruder cannot delay, duplicate, remove messages, or write messages onto this channel under a fake identity (unless he has corrupted a party). However, just as in [5], the intruder can read the messages written onto the secure channel. We note that our results also hold in case of read-protected secure channels. Another difference to standard Dolev-Yao models is that, in order to be able to formulate game-theoretic properties, we explicitly describe the behavior of a protocol as an infinite-state transition graph which comprises all runs of a protocol.

We now describe the model in more detail by defining terms and messages, the intruder, principals and protocols, and the transition graph.

Terms and Messages. As usual, we have a finite set $\mathcal{V}$ of variables, a finite set $\mathcal{A}$ of atoms, a finite set $\mathcal{K}$ of public and private keys equipped with a bijection $\cdot^{-1}$ assigning public to private keys and vice versa. In addition, we have a finite set $\mathcal{N}$ of *principal addresses* for the secure channels and an *infinite* set $\mathcal{A}_I$ of *intruder atoms*, containing nonces and symmetric keys the intruder can generate. All of the mentioned sets are assumed to be disjoint.

We define two kinds of terms by the following grammar, namely *plain terms* and *secure channel terms*:

$$\begin{aligned} \textit{plain-terms} &::= \mathcal{V} \mid \mathcal{A} \mid \mathcal{A}_I \mid \langle \textit{plain-terms}, \textit{plain-terms} \rangle \mid \{\textit{plain-terms}\}^s_{\textit{plain-terms}} \mid \\ &\quad \{\textit{plain-terms}\}^a_{\mathcal{K}} \mid \mathsf{hash}(\textit{plain-terms}) \mid \mathsf{sig}_{\mathcal{K}}(\textit{plain-terms}) \\ \textit{sec-terms} &::= \mathsf{sc}(\mathcal{N}, \mathcal{N}, \textit{plain-terms}) \\ \textit{terms} &::= \textit{plain-terms} \mid \textit{sec-terms} \mid \mathcal{N} \end{aligned}$$

While the plain terms are standard in Dolev-Yao models, a secure channel term of the form $\mathsf{sc}(n, n', t)$ stands for feeding the secure channel from n to n' with t. Knowing n grants access to secure channels with sender address n. A *(plain/secure channel) message* is a (plain/secure channel) ground term, i.e., a term without variables.

Intruder. Given a set $\mathcal{I}$ of messages, the (infinite) set $d(\mathcal{I})$ of messages the intruder can derive from $\mathcal{I}$ is the smallest set satisfying the following conditions: $\mathcal{I} \subseteq d(\mathcal{I})$; if $m, m' \in d(\mathcal{I})$, then $\langle m, m' \rangle \in d(\mathcal{I})$; if $\langle m, m' \rangle \in d(\mathcal{I})$, then $m \in d(\mathcal{I})$ and $m' \in d(\mathcal{I})$; if $m, m' \in d(\mathcal{I})$, then $\{m\}^s_{m'} \in d(\mathcal{I})$; if $\{m\}^s_{m'} \in d(\mathcal{I})$ and $m' \in d(\mathcal{I})$, then $m \in d(\mathcal{I})$; if $m \in d(\mathcal{I})$ and $k \in d(\mathcal{I}) \cap \mathcal{K}$, then $\{m\}^a_k \in d(\mathcal{I})$; if $\{m\}^a_k \in d(\mathcal{I})$ and $k^{-1} \in d(\mathcal{I})$, then $m \in d(\mathcal{I})$; if $m \in d(\mathcal{I})$, then $\mathsf{hash}(m) \in d(\mathcal{I})$; if $m \in d(\mathcal{I})$ and $k^{-1} \in d(\mathcal{I}) \cap \mathcal{K}$, then $\mathsf{sig}_k(m)$ (the signature contains the public key but can only be generated if the corresponding private key is known); if $m \in d(\mathcal{I})$, $n \in d(\mathcal{I}) \cap \mathcal{N}$, and $n' \in \mathcal{N}$, then $\mathsf{sc}(n, n', m) \in d(\mathcal{I})$ (*writing onto the secure channel*); $\mathcal{A}_I \subseteq d(\mathcal{I})$ (*generating fresh constants*).

Intuitively, $n \in d(\mathcal{I}) \cap \mathcal{N}$ means that the intruder has corrupted the principal with address n and therefore can impersonate this principal when writing onto the secure channel.

In our model, all (strongly) dishonest parties are subsumed in the intruder. Weakly dishonest parties can be modeled as principals whose specification deviates from the specification of the protocol.

Principals and Protocols. *Principal rules* are of the form $R \Rightarrow S$ where R is a term or ε and S is a term.

A *rule tree* $\Pi = (V, E, r, \ell)$ is a finite tree rooted at $r \in V$ where ℓ maps every edge $(v, v') \in E$ of Π to a principal rule $\ell(v, v')$.

A *principal* is a tuple consisting of a rule tree $\Pi = (V, E, r, \ell)$ and a finite set of plain messages, the *initial knowledge of the principal*. We require that every variable occurring on the right-hand side of a principal rule $\ell(v, v')$ in Π also occurs on the left-hand side of $\ell(v, v')$ or on the left-hand side of a principal rule on the path from r to v. In addition, and unlike [11], we require a condition necessary for the principal to perform a feasible computation: The decryption and signature verification operations performed when receiving a message can actually be carried out, i.e., terms in key positions (t' in $\{t\}^s_{t'}$, k^{-1} in $\{t\}^a_k$, and k in $\mathsf{sig}_k(t)$) on the left-hand side of principal rules can be derived from the set consisting of the left-hand side of the current principal rule, the left-hand sides of preceeding rules, and the initial knowledge of the principal. Obviously, the above condition is satisfied for all realistic principals. Moreover, it allows to simplify the constraint-based algorithm (Section 5).

For $v \in V$, we write $\Pi{\downarrow}v$ to denote the subtree of Π rooted at v. For a substitution σ, we write $\Pi\sigma$ for the principal obtained from Π by substituting all variables x occurring in the principal rules of Π by $\sigma(x)$.

A *protocol* $P = ((\Pi_1, \ldots, \Pi_n), \mathcal{I})$ consists of a finite sequence of principals Π_i and a finite set $\mathcal{I}$ of messages, the *initial intruder knowledge*. We require that each variable occurs in the rules of only one principal, i.e., different principals must have disjoint sets of variables. We assume that intruder atoms, i.e., elements of $\mathcal{A}_I$, do not occur in P.

As an example protocol, let us consider P_{ex} as depicted in Figure 1. This protocols consists of two principals Π_1 and Π_2 and the initial knowledge $\mathcal{I}_0 = \{\{a\}^s_k, \{b\}^s_k\}$ of the intruder. The principals are described by the two trees on the left-hand side of the figure; the tree on the right-hand side is used later. Informally speaking, Π_2 can, without waiting for input from the secure channel or the intruder, decide whether to write $\langle a, b\rangle$ or $\langle b, b\rangle$ into the secure channel from Π_2 to Π_1. While the intruder can read the message written into this channel, he cannot modify or delay this message. Also, he cannot insert his own message into this channel as he does not have the principal address 2 in his intruder knowledge, and hence, cannot generate messages of the form $\mathsf{sc}(2, \cdot, t)$. Consequently, such messages must come from Π_2. Principal Π_1 first waits for a message of the form $\langle x, b\rangle$ in the secure channel from Π_2 to Π_1. In case Π_2 wrote, say, $\langle a, b\rangle$ into this channel, x is substituted by a, and this message is written into the network, and hence, given to the intruder. Next, Π_1 waits for input of the form $\{y\}^s_k$. This is not a secure channel term, and thus, comes from the intruder. In case the intruder sends $\{b\}^s_k$, say, then y is substituted by b. Finally, Π_1 waits for input of the form a (in the edges from f_3 to f_4 and f_3 to f_5) or b (in the edge from f_3 to f_6). Recall that x was substituted by a and y by b. If the intruder sends b, say, then Π_2 takes the edge from f_3 to f_6 and outputs c_2 into the network. If the intruder had sent a, Π_1 could have chosen between the first two edges. We note that this protocol is not meant to perform a useful task. It is rather used to illustrate different aspects of our constraint-based algorithm ([11] contains a formal specification of the contract-signing protocol by Asokan, Shoup, and Waidner [3] in our model).

2.1 Transition Graph Induced by a Protocol

A transition graph $\mathcal{G}_P$ induced by a protocol P comprises all runs of a protocol. To define this graph, we first introduce states and transitions between these states.

A *state* is of the form $((\Pi_1, \ldots, \Pi_n), \sigma, \mathcal{I}, \mathcal{S})$ where σ is a ground substitution, for each i, Π_i is a rule tree such that $\Pi_i\sigma$ is a principal, $\mathcal{I}$ is a finite set of messages, the *intruder knowledge*, and $\mathcal{S}$ is a finite multi-set of secure channel messages, the *secure channel*. The idea is that when the transition system gets to such a state, then the

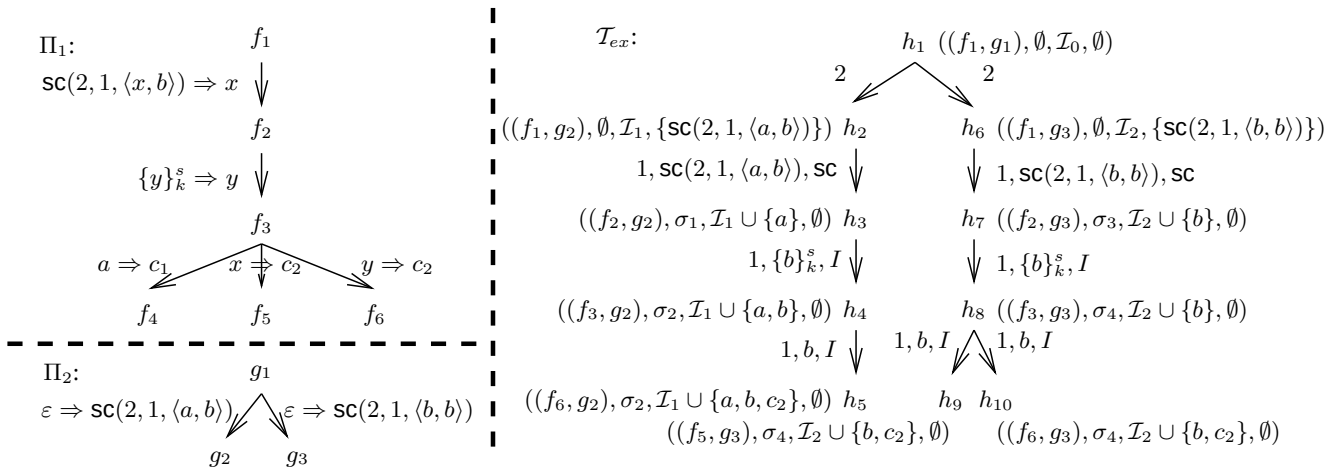


Figure 1: Protocol $P_{ex} = (\{\Pi_1,\Pi_2\},\mathcal{I}_0)$ with $\mathcal{I}_0 = \{\{a\}^s_k,\{b\}^s_k\}$, initial knowledge $\{1,a,b,k,c_1,c_2\}$ of Π_1 and initial knowledge $\{2,a,b\}$ of Π_2. Strategy tree $\mathcal{T}_{ex}$ for P_{ex} with $\mathcal{I}_1 = \mathcal{I}_0 \cup \{\langle a,b\rangle\}$, $\mathcal{I}_2 = \mathcal{I}_0 \cup \{\langle b,b\rangle\}$, $\sigma_1 = \{x \mapsto a\}$, $\sigma_2 = \sigma_1 \cup \{y \mapsto b\}$, $\sigma_3 = \{x \mapsto b\}$, and $\sigma_4 = \sigma_3 \cup \{y \mapsto b\}$. Also, for brevity of notation, in the first component of the states we write, for instance, f_1 instead of $\Pi_1{\downarrow}f_1$. The strategy property we consider is $((C_{ex},C'_{ex})) = ((\{c_2\},\{c_1\}))$.

substitution σ has been performed, the accumulated intruder knowledge is what can be derived from $\mathcal{I}$, the secure channels hold the messages in $\mathcal{S}$, and for each i, Π_i is the "remaining protocol" to be carried out by principal i. This also explains why $\mathcal{S}$ is a multi-set: messages sent several times should be delivered several times. Given a protocol $P = ((\Pi_1,\ldots,\Pi_n),\mathcal{I})$ the *initial state of P* is $((\Pi_1,\ldots,\Pi_n),\sigma,\mathcal{I},\emptyset)$ where σ is the substitution with empty domain.

We have three kinds of transitions: intruder, secure channel, and ε-transitions. In what follows, let $\Pi_i = (V_i,E_i,r_i,\ell_i)$ and $\Pi'_i = (V'_i,E'_i,r'_i,\ell'_i)$ denote rule trees. We define under which circumstances there is a transition

$$((\Pi_1,\ldots,\Pi_n),\sigma,\mathcal{I},\mathcal{S}) \xrightarrow{\tau} ((\Pi'_1,\ldots,\Pi'_n),\sigma',\mathcal{I}',\mathcal{S}') \tag{1}$$

with τ an appropriate label.

1. *Intruder transitions:* The transition (1) with label i,m,I exists if there exists $v \in V_i$ with $(r_i,v) \in E_i$ and $\ell_i(r_i,v) = R \Rightarrow S$, and a substitution σ'' of the variables in $R\sigma$ such that (a) $m \in d(\mathcal{I})$, (b) $\sigma' = \sigma \cup \sigma''$, (c) $R\sigma' = m$, (d) $\Pi'_j = \Pi_j$ for every $j \neq i$, $\Pi'_i = \Pi_i{\downarrow}v$, (e) $\mathcal{I}' = \mathcal{I} \cup \{S\sigma'\}$ if $S \neq \mathsf{sc}(\cdot,\cdot,\cdot)$, and $\mathcal{I}' = \mathcal{I} \cup \{t\sigma'\}$ if $S = \mathsf{sc}(\cdot,\cdot,t)$ for some t, (f) $\mathcal{S}' = \mathcal{S}$ if $S \neq \mathsf{sc}(\cdot,\cdot,\cdot)$, and $\mathcal{S}' = \mathcal{S} \cup \{S\sigma'\}$ otherwise. This transition models that principal i reads the message m from the intruder (i.e., the public network).
2. *Secure channel transitions:* The transition (1) with label i,m,sc exists if there exists $v \in V_i$ with $(r_i,v) \in E_i$ and $\ell_i(r_i,v) = R \Rightarrow S$, and a substitution σ'' of the variables in $R\sigma$ such that $m \in \mathcal{S}$, (b)–(e) from 1., and $\mathcal{S}' = \mathcal{S} \setminus \{m\}$ if $S \neq \mathsf{sc}(\cdot,\cdot,\cdot)$, and $\mathcal{S}' = (\mathcal{S} \setminus \{m\}) \cup \{S\sigma'\}$ otherwise. This transition models that principal i reads message m from the secure channel.
3. *ε-transitions:* The transition (1) with label i exists if there exists $v \in V_i$ with $(r_i,v) \in E_i$ and $\ell_i(r_i,v) = \varepsilon \Rightarrow S$ such that $\sigma' = \sigma$ and (d), (e), (f) from above. This transition models that i performs a step where neither a message is read from the intruder nor from the secure channel.

Given a protocol P, the *transition graph $\mathcal{G}_P$ induced by P* is the tuple (S_P,E_P,q_P) where q_P is the initial state of P, S_P is the set of states reachable from q_P by a sequence of transitions, and E_P is the set of all transitions among states in S_P. We write $q \in \mathcal{G}_P$ if q is a state in $\mathcal{G}_P$ and $q \xrightarrow{\tau} q' \in \mathcal{G}_P$ if $q \xrightarrow{\tau} q'$ is a transition in $\mathcal{G}_P$.

We note that $\mathcal{G}_P$ is a DAG since by performing a transition, the size of the first component of a state decreases. While the graph may be infinite branching, the maximal length of a path in this graph is bounded by the total number of edges in the principals Π_i of P.

3 Intruder Strategies and Strategy Properties

We now define intruder strategies on transition graphs and the goal the intruder tries to achieve following his strategy. As mentioned in the introduction, while in [11] positional intruder strategies have been considered where the strategy of the intruder at a global state of the protocol run may only depend on the current state, here we allow the intruder to take the whole path leading to this state (i.e., the history) into account. While this potentially provides the intruder with more power, these notions are in fact equivalent (see Proposition 2). The main motivation for the new notion of strategy is that it is much better suited for the constraint solving approach (Section 5).

To define intruder strategies, we introduce the notion of a strategy tree, which captures that the intruder has a way of acting such that regardless of how the other principals act he achieves a certain goal, where goal in our context means that a state will be reached where the intruder can derive certain constants and cannot derive others (e.g., for balance, the intruder tries to obtain IntruderHasContract but tries to prevent HonestPartyHasContract from occurring).

More concretely, let us consider the protocol P_{ex} depicted in Figure 1. We want to know if the intruder has a strategy to get to a state where he can derive atom c_2 but not atom c_1 (no matter what the principals Π_1 and Π_2 do). Such a strategy of the intruder has to deal with both decisions principal Π_2 may make in the first step because the intruder cannot control which edge is taken by Π_2. It turns out that regardless of which message is sent by principal Π_2 in its first step, the following simple strategy allows the intruder to achieve his goal: The intruder can send $\{b\}_k^s$ to principal Π_1 in the second step of Π_1 and in the last step of Π_1, the intruder sends b to principal Π_1. This guarantees that in the last step of Π_1, the left-most edge is never taken, and thus, c_1 is not returned, but at least one of the other two edges can be taken, which in any case yields c_2. Formally, such strategies are defined as trees. In our example, the strategy tree corresponding to the strategy informally explained above is depicted on the right-hand side of Figure 1.

Definition 1 *For $q \in \mathcal{G}_P$ a q-*strategy tree *$\mathcal{T}_q = (V, E, r, \ell_V, \ell_E)$ is an unordered tree where every vertex $v \in V$ is mapped to a state $\ell_V(v) \in \mathcal{G}_P$ and every edge $(v, v') \in E$ is mapped to a label of a transition such that the following conditions are satisfied for all $v, v' \in V$, principals j, messages m, and states q', q'':*

1. $\ell_V(r) = q$.
2. $\ell_V(v) \xrightarrow{\ell_E(v,v')} \ell_V(v') \in \mathcal{G}_P$ *for all* $(v, v') \in E$. *(Edges correspond to transitions.)*
3. *If* $\ell_V(v) = q'$ *and* $q' \xrightarrow{j} q'' \in \mathcal{G}_P$, *then there exists* $v'' \in V$ *such that* $(v, v'') \in E$, $\ell_V(v'') = q''$, *and* $\ell_E(v, v'') = j$. *(All ε-transitions originating in q' must be present in $\mathcal{T}_q$.)*
4. *If* $\ell_V(v) = q'$ *and* $q' \xrightarrow{j,m,\mathsf{sc}} q'' \in \mathcal{G}_P$, *then there exists* $v'' \in V$ *such that* $(v, v'') \in E$, $\ell_V(v'') = q''$, *and* $\ell_E(v, v'') = j, m, \mathsf{sc}$. *(The same as 3. for secure channel transitions.)*
5. *If* $(v, v') \in E$, $\ell_E(v, v') = j, m, I$, *and there exists* $q'' \neq \ell_V(v')$ *with* $\ell_V(v) \xrightarrow{j,m,I} q'' \in \mathcal{G}_P$, *then there exists* v'' *with* $(v, v'') \in E$, $\ell_E(v, v'') = j, m, I$ *and* $\ell_V(v'') = q''$. *(The intruder cannot choose which principal rule is taken by j if several are possible given the input provided by the intruder.)*

A *strategy property*, i.e., the goal the intruder tries to achieve, is a tuple $((C_1, C_1'), \ldots, (C_l, C_l'))$ where $C_i, C_i' \subseteq \mathcal{A} \cup \mathcal{K} \cup \mathcal{N}$. A state $q \in \mathcal{G}_P$ satisfies $((C_1, C_1'), \ldots, (C_l, C_l'))$ if there exist q-strategy trees $\mathcal{T}_1, \ldots, \mathcal{T}_l$ such that

every $\mathcal{T}_i$ satisfies (C_i, C_i') where $\mathcal{T}_i$ satisfies (C_i, C_i') if for all leaves v of $\mathcal{T}_i$ all elements from C_i can be derived by the intruder and all elements from C_i' cannot, i.e., $C_i \subseteq d(\mathcal{I})$ and $C_i' \cap d(\mathcal{I}) = \emptyset$ where $\mathcal{I}$ denotes the intruder knowledge in state $\ell_V(v)$.

The decision problem G-STRATEGY *(with general rather than positional intruder strategy)* asks, given a protocol P and a strategy property $((C_1, C_1'), \ldots, (C_l, C_l'))$, whether there exists a state $q \in \mathcal{G}_P$ that satisfies the property. In this case we write $(P, (C_1, C_1'), \ldots, (C_l, C_l')) \in$ G-STRATEGY.

Note that in a q-strategy tree $\mathcal{T}_q$ there may exist vertices $v' \neq v$ with $\ell_V(v') = \ell_V(v)$ such that the subtrees $\mathcal{T}_q{\downarrow}v$ and $\mathcal{T}_q{\downarrow}v'$ of $\mathcal{T}_q$ rooted at v and v', respectively, are not isomorphic. In other words, the intruder's strategy may depend on the path that leads to a state (i.e., the history) rather than on the state alone, as is the case for positional strategies. As mentioned, the strategies defined in [11] are positional. Let P-STRATEGY be defined analogously to G-STRATEGY with positional intruder strategies (see [11] for the precise definition). Using that our strategy properties only constrain the leaves of strategy trees, the following is not hard to show.

Proposition 2 $(P, (C_1, C_1'), \ldots, (C_l, C_l')) \in$ G-STRATEGY *iff* $(P, (C_1, C_1'), \ldots, (C_l, C_l')) \in$ P-STRATEGY.

In [11] it was shown that P-STRATEGY is decidable. As an immediate corollary of Proposition 2 we obtain:

Corollary 3 G-STRATEGY *is decidable.*

4 Constraint Solving

In this section, we introduce constraint systems and state the well-known fact that procedures for solving these systems exist (see, e.g., [13] for more details). In Section 5, we will then use such a procedure as a black-box for our constraint-based algorithm.

A *constraint* is of the form $t : T$ where t is a plain term and T is a finite non-empty set of plain terms. Since we will take care of secure channel terms when turning the symbolic branching structure into a constraint system, we can disallow secure channel terms in constraints.

A *constraint system* $\mathbf{C}$ is a tuple consisting of a sequence $s = t_1 : T_1, \ldots, t_n : T_n$ of constraints and a substitution τ such that the domain of τ is disjoint from the set of variables occurring in s and, for all x in the domain of τ, $\tau(x)$ only contains variables also occurring in s. We call $\mathbf{C}$ *simple* if t_i is a variable for all i. We call $\mathbf{C}$ *valid* if it satisfies the origination and monotonicity property as defined in [13]. These are standard restrictions on constraint systems imposed by constraint solving procedures. Valid constraint systems are all that is needed in our setting.

A ground substitution σ where the domain of σ is the set of variables in $t_1 : T_1, \ldots, t_n : T_n$ is a *solution* of $\mathbf{C}$ ($\sigma \vdash \mathbf{C}$) if $t_i\sigma \in d(T_i\sigma)$ for every i. We call $\sigma \circ \tau$ (the composition of σ and τ read from right to left) a *complete* solution of $\mathbf{C}$ ($\sigma \circ \tau \vdash_c \mathbf{C}$) with τ as above.

A simple constraint system $\mathbf{C}$ obviously has a solution. One such solution, which we denote by σ_C, replaces all variables in $\mathbf{C}$ by new intruder atoms $a \in \mathcal{A}_I$ where different variables are replaced by different atoms. We call σ_C the *solution associated with* $\mathbf{C}$ and $\sigma_C \circ \tau$ the *complete solution associated with* $\mathbf{C}$.

Given a constraint system $\mathbf{C}$, a finite set $\{\mathbf{C}_1, \ldots, \mathbf{C}_n\}$ of simple constraint systems is called a *sound and complete solution set for* $\mathbf{C}$ if $\{\nu \mid \nu \vdash_c \mathbf{C}\} = \{\nu \mid \exists i \text{ s.t. } \nu \vdash_c \mathbf{C}_i\}$. Note that $\mathbf{C}$ does not have a solution iff $n = 0$.

The following fact is well-known (see, e.g., [7, 13, 4] and references therein):

Fact 1 *There exists a procedure which given a valid constraint system* $\mathbf{C}$ *outputs a sound and complete solution set for* $\mathbf{C}$.

While different constraint solving procedures (and implementations thereof) may compute different sound and complete solution sets, our constraint-based algorithm introduced in Section 5 works with any of these procedures.

It is only important that the set computed is sound and complete. As already mentioned in the introduction, to decide reachability properties it suffices if the procedure only returns one simple constraint system in the sound and complete set. However, the constraint solving procedures proposed in the literature are typically capable of returning a sound and complete solution set.

In what follows, we fix one such procedure and call it the *constraint solver*. More precisely, w.l.o.g., we consider the constraint solver to be a non-deterministic algorithm which non-deterministically chooses a simple constraint system from the sound and complete solution set and returns this system as output. We require that for every simple constraint system in the sound and complete solution set, there is a run of the constraint solver that returns this system. If the sound and complete set is empty, the constraint solver always returns no.

We note that while standard constraint solving procedures can deal with the cryptographic primitives considered here, these procedures might need to be extended when adding further cryptographic primitives. For example, this is the case for private contract signatures, which are used in some contract signing protocols [9] and were taken into account in [11]. However, constraint solving procedures can easily be extended to deal with these signatures. We have not considered them here for brevity of presentation and since the main focus of the present work is not to extend constraint solving procedures but to show how these procedures can be employed to deal with game-theoretic security properties.

5 The Constraint-Based Algorithm

We now present our constraint-based algorithm, called SolveStrategy, for deciding G-STRATEGY. As mentioned, it uses a standard constraint solver (Fact 1) as a subprocedure.

In what follows, we present the main steps performed by SolveStrategy, with more details given in subsequent sections. The input to SolveStrategy is a protocol P and a strategy property $((C_1, C_1'), \ldots, (C_l, C_l'))$.

1. Guess a symbolic branching structure $\mathbf{B}$, i.e., guess a symbolic path π^s from the initial state of P to a symbolic state q^s and a symbolic q^s-strategy tree $\mathcal{T}^s_{i,q^s}$ for every (C_i, C_i') starting from this state (see Section 5.1 for more details).
2. Derive from $\mathbf{B} = \pi^s, \mathcal{T}^s_{1,q^s}, \ldots, \mathcal{T}^s_{l,q^s}$ and the strategy property $((C_1, C_1'), \ldots, (C_l, C_l'))$ the induced and valid (!) constraint system $\mathbf{C} = \mathbf{C_B}$ (see Section 5.2 for the definition). Then, run the constraint solver on $\mathbf{C}$. If it returns no, then halt. Otherwise, let $\mathbf{C}'$ be the simple constraint system returned by the solver. (Recall that $\mathbf{C}'$ belongs to the sound and complete solution set and is chosen non-deterministically by the solver.)
3. Let ν be the complete solution associated with $\mathbf{C}'$. Check whether ν when applied to $\mathbf{B}$ yields a valid path in $\mathcal{G}_P$ from the initial state of P to a state q and q-strategy trees $\mathcal{T}_{i,q}$ satisfying (C_i, C_i') for every i. If so, output yes and $\mathbf{B}$ with ν applied, and otherwise return no (see Section 5.3 for more details). In case yes is returned, $\mathbf{B}$ with ν applied yields a concrete solution of the problem instance $(P, (C_1, C_1'), \ldots, (C_l, C_l'))$.

We emphasize that, for simplicity of presentation, SolveStrategy is formulated as a non-deterministic algorithm. Hence, the overall decision of SolveStrategy is yes if there exists at least one computation path where yes is returned. Otherwise, the overall decision is no (i.e., $(P, (C_1, C_1'), \ldots, (C_l, C_l')) \notin$ G-STRATEGY).

In the following three sections, the three steps of SolveStrategy are further explained. Our main result is the following theorem:

Theorem 4 *SolveStrategy is a decision procedure for* G-STRATEGY.

While we know that the problem G-STRATEGY is decidable from Corollary 3, the main point of Theorem 4 is that SolveStrategy uses standard constraint solving procedures as a black-box, and as such, is a good basis for extending existing practical constraint-based algorithms for reachability properties to deal with game-theoretic security properties.

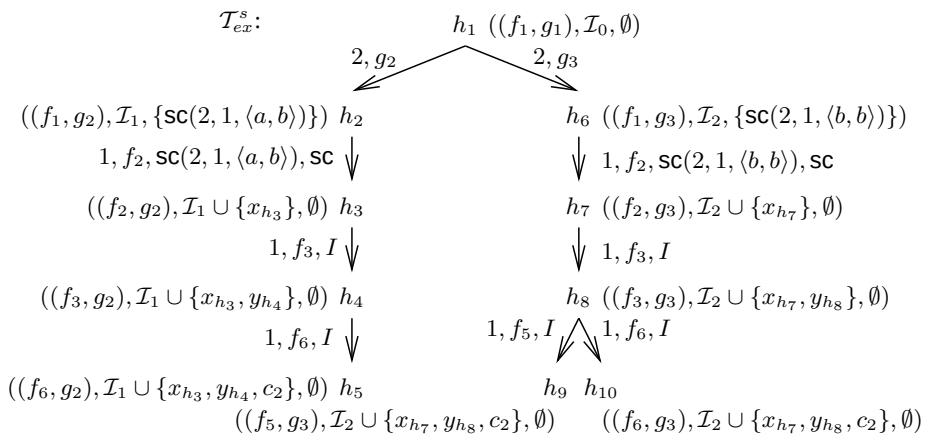


Figure 2: Symbolic strategy tree $\mathcal{T}^s_{ex}$ for the protocol P_{ex} where $\mathcal{I}_1 = \mathcal{I}_0 \cup \{\langle a, b\rangle\}$ and $\mathcal{I}_2 = \mathcal{I}_0 \cup \{\langle b, b\rangle\}$. For brevity of notation, in the first component of the symbolic states we write, for instance, f_1 instead of $\Pi_1{\downarrow}f_1$.

The proof of Theorem 4 is quite different from the cut-and-paste argument in [11] where, similar to [14], it was shown that an attack can be turned into a "small" attack. Here we rather make use of the fact that procedures for computing sound and complete solution sets exist, which makes our proof (and also our algorithm) more modular and easier to extend.

We note that if we used positional strategies as in [11], SolveStrategy would have to be extended to guess the symbolic states of symbolic branching structures that coincide after the substitution ν is applied. To avoid this, we employ the strategies with history as explained above.

5.1 Guess the Symbolic Branching Structure

To describe the first step of SolveStrategy in more detail, we first define symbolic branching structures, which consist of symbolic paths and symbolic strategy trees. To define symbolic paths and strategy trees, we need to introduce symbolic states, transitions, and trees (see [10] for full details). These notions will be illustrated by the example in Figure 1

A *symbolic state* $q^s = ((\Pi_1, \ldots, \Pi_n), \mathcal{I}, \mathcal{S})$ is defined just as a concrete state (see Section 2.1) except that the substitution is omitted and the intruder knowledge $\mathcal{I}$ and the secure channel $\mathcal{S}$ may contain terms (with variables) instead of only messages. The *symbolic initial state* of a protocol $P = ((\Pi_1, \ldots, \Pi_n), \mathcal{I}_0)$ is $((\Pi_1, \ldots, \Pi_n), \mathcal{I}_0, \emptyset)$.

A *symbolic transition*, analogously to concrete transitions, is a transition between symbolic states and is of the form

$$((\Pi_1, \ldots, \Pi_n), \mathcal{I}, \mathcal{S}) \xrightarrow{\ell} ((\Pi'_1, \ldots, \Pi'_n), \mathcal{I}', \mathcal{S}') \tag{2}$$

with ℓ an appropriate label where again we distinguish between symbolic intruder, secure channel, and ε-transitions. Informally speaking, these transitions are of the following form (see [10] for details and the example below): For *symbolic intruder transitions* the label ℓ is of the form i, f, I where now f is not the message delivered by the intruder, as was the case for concrete intruder transitions, but a direct successor of the root r_i of Π_i. The intuition is that the principal rule $R \Rightarrow S$ the edge (r_i, f) is labeled with in Π_i is applied. The symbolic state $((\Pi_1, \ldots, \Pi_n), \mathcal{I}, \mathcal{S})$ is updated accordingly to $((\Pi'_1, \ldots, \Pi'_n), \mathcal{I}', \mathcal{S}')$ (see the example below). We call $R \Rightarrow S$ the *principal rule associated with the symbolic transition*. Similarly, the label of a *symbolic secure channel transition* is of the form i, f, R', sc where f is interpreted as before and R' is the term read from the secure channel. If $R \Rightarrow S$ is the principal rule associated with the transition, then $\mathcal{S}'$ is obtained by removing R' from $\mathcal{S}$

and adding S if S is a secure channel term. When constructing the constraint system, we will guarantee that R' unifies with R. Finally, the label of *symbolic ε-transitions* is of the form i, f with the obvious meaning.

A *symbolic q^s-tree* $\mathcal{T}^s_{q^s} = (V, E, r, \ell_V, \ell_E)$ is an unordered finite tree where the vertices are labeled with symbolic states, the root is labeled with q^s, and the edges are labeled with labels of symbolic transitions such that an edge (v, v') of the tree, more precisely, the labels of v and v' and the label of (v, v') correspond to symbolic transitions. We call the principal rule associated with such a symbolic transition *the principal rule associated with* (v, v'). Note that the symbolic transitions of different edges may be associated with the same principal rule. Now, since the same rule may occur at different positions in the tree, its variables may later be substituted differently. We therefore need a mechanism to consistently rename variables.

Figure 2 depicts a symbolic q^s_0-tree $\mathcal{T}^s_{ex}$ for P_{ex} (Figure 1) where $q^s_0 = (\{\Pi_1, \Pi_2\}, \mathcal{I}_0, \emptyset)$ is the symbolic initial state of P_{ex}. For brevity of notation, just as in the case of the strategy tree in Figure 1, the first component of the symbolic states in this tree does not contain the principals but only their corresponding roots. Note that the principal rules of Π_1 are applied at different places in this tree. Therefore, different copies of the variables x and y need to be introduced, which we do by indexing the variables by the name of the vertex where the rule is applied. This yields the variables $x_{h_3}, x_{h_7}, y_{h_4}, y_{h_8}$ in $\mathcal{T}^s_{ex}$.

A *symbolic path* π^s of a protocol P is a symbolic q^s_0-tree where every vertex has at most one successor and q^s_0 is the symbolic initial state of P.

A *symbolic q^s-strategy tree* $\mathcal{T}^s_{q^s} = (V, E, r, \ell_V, \ell_E)$ is a symbolic q^s-tree which satisfies additional conditions. Among others, we require that in one node of this tree the intruder may only send a message to one principal Π_i; we show that this is w.l.o.g. Also, all ε-transitions applicable in one node are present. Symbolic strategy trees are defined in such a way that for every symbolic state q^s the number of symbolic q^s-strategy trees is finite and all such trees can effectively be generated. The tree depicted in Figure 2 is a symbolic q^s_0-strategy tree.

For a protocol P and strategy property $((C_1, C'_1), \ldots, (C_l, C'_l))$, a *symbolic branching structure* is of the form $\mathbf{B}^s = \pi^s, \mathcal{T}^s_1, \ldots, \mathcal{T}^s_l$ where π^s is a symbolic path of P and the $\mathcal{T}^s_i$ are symbolic q^s-strategy trees where q^s is the symbolic state the leaf of π^s is labeled with. Given a protocol and a strategy property, there are only a finite number of symbolic branching structures and these structures can be generated by an algorithm. In particular, there is a non-deterministic algorithm which can guess one symbolic branching structure $\mathbf{B}^s$ among all possible such structures.

For the strategy property $((C_{ex}, C'_{ex})) = ((\{c_2\}, \{c_1\}))$, we can consider $\mathcal{T}^s_{ex}$ in Figure 2 also as a symbolic branching structure $\mathbf{B}^s_{ex}$ of P_{ex} where the path π^s is empty and $l = 1$.

5.2 Construct and Solve the Induced Constraint System

We now show how the constraint system $\mathbf{C} = \mathbf{C_B}$ is derived from the symbolic branching structure $\mathbf{B} = \pi^s, \mathcal{T}^s_1, \ldots, \mathcal{T}^s_l$ (guessed in the first step of SolveStrategy) and the given strategy property $((C_1, C'_1), \ldots, (C_l, C'_l))$. This constraint system can be shown to be valid, and hence, by Fact 1, a constraint solver can be used to solve it. In this extended abstract, we only illustrate how **C** is derived from **B** and the strategy property by the example in Figure 1 (see [10] for full definitions).

Before turning to the example, we informally explain how to encode in a constraint system communication involving the secure channel. (Another, somewhat less interesting issue is how to deal with secure channel terms generated by the intruder. This is explained in our technical report [10].) The basic idea is that we write messages intended for the secure channel into the intruder's knowledge and let the intruder deliver these messages. The problem is that while every message in the secure channel can only be read once, the intruder could try to deliver the same message several times. To prevent this, every such message when written into the intruder's knowledge is encrypted with a *new* key not known to the intruder and this key is also (and only) used in the principal rule which according to the symbolic branching structure is supposed to read the message. This guarantees that the intruder cannot abusively deliver the same message several times to unintended recipients or make use of these

encrypted messages in other contexts. Here we use the restriction on principals introduced in Section 2, namely that decryption keys can be derived by a principal. Without this condition, a principal rule of the form $\{y\}^s_x \Rightarrow x$ would be allowed even if the principal does not know (i.e., cannot derive) x. Such a rule would equip a principal with the unrealistic ability to derive any secret key from a ciphertext. Hence, the intruder, using this principal as an oracle, could achieve this as well and could potentially obtain the new keys used to encrypt messages intended for the secure channel.

We now turn to our example and explain how the (valid) constraint system, called $\mathbf{C}_{ex}$, derived from $\mathbf{B}^s_{ex}$ and $((C_{ex}, C'_{ex}))$ looks like and how it is derived from $\mathbf{B}^s_{ex}$, where $\mathbf{B}^s_{ex}$, as explained above, is simply the symbolic strategy tree $\mathcal{T}^s_{ex}$ (Figure 2): $\mathbf{C}_{ex}$ is the following sequence of constraints with an empty substitution where $k_1, k_2, k_3 \in \mathcal{A}$ are new atoms and we write $t_1, \ldots, t_n$ instead of $\{t_1, \ldots, t_n\}$.

$$
\begin{array}{rrcl}
1. & \{\langle x_{h_3}, b\rangle\}^s_{k_1} & : & \mathcal{I}_1, \{\langle a, b\rangle\}^s_{k_1} \\
2. & \{\langle x_{h_7}, b\rangle\}^s_{k_2} & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2} \\
3. & \{y_{h_4}\}^s_k & : & \mathcal{I}_1, \{\langle a, b\rangle\}^s_{k_1}, x_{h_3} \\
4. & \{y_{h_8}\}^s_k & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2}, x_{h_7} \\
5. & y_{h_4} & : & \mathcal{I}_1, \{\langle a, b\rangle\}^s_{k_1}, x_{h_3}, y_{h_4}
\end{array}
\qquad
\begin{array}{rrcl}
6. & x_{h_7} & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2}, x_{h_7}, y_{h_8} \\
7. & y_{h_8} & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2}, x_{h_7}, y_{h_8} \\
8. & c_2 & : & \mathcal{I}_1, \{\langle a, b\rangle\}^s_{k_1}, x_{h_3}, y_{h_4}, c_2 \\
9. & c_2 & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2}, x_{h_7}, y_{h_8}, c_2 \\
10. & c_2 & : & \mathcal{I}_2, \{\langle b, b\rangle\}^s_{k_2}, x_{h_7}, y_{h_8}, c_2
\end{array}
$$

This constraint system is obtained from $\mathbf{B}^s_{ex}$ as follows: We traverse the vertices of $\mathbf{B}^s_{ex}$ in a top-down breadth first manner. Every edge induces a constraint except those edges which correspond to symbolic ε-transitions. This is how the constraints 1.–7. come about where 1., 3., and 5. are derived from the left branch of $\mathbf{B}^s_{ex}$ and 2., 4., 6., and 7. from the right branch. Note that in 1. and 2. we encode the communication with the secure channel by encrypting the terms with new keys k_1 and k_2. The terms $\{\langle a, b\rangle\}^s_{k_1}$ and $\{\langle b, b\rangle\}^s_{k_2}$ are not removed anymore from the right-hand side of the constraints, i.e., from the intruder knowledge, in order for $\mathbf{C}_{ex}$ to satisfy the monotonicity property of constraint systems (recall that monotonicity is necessary for the validity of constraint systems). As explained above, since we use *new* keys and due to the restriction on principals, this does not cause problems. The constraints 8.–10. are used to ensure that c_2 can be derived at every leaf of $\mathcal{T}^s_{ex}$, a requirement that comes from our example security property $((C_{ex}, C'_{ex}))$ where $C_{ex} = \{c_2\}$. In vertex h_8 of $\mathcal{T}^s_{ex}$, two symbolic intruder transitions leave the vertex, which, as explained above, means that the associated principal rules should both be able to read the message delivered by the intruder.

Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be constraint systems with empty sequences of constraints and the substitution $\nu_1 = \{x_{h_3} \mapsto a, x_{h_7} \mapsto b, y_{h_4} \mapsto a, y_{h_8} \mapsto b\}$ and $\nu_2 = \{x_{h_3} \mapsto a, x_{h_7} \mapsto b, y_{h_4} \mapsto b, y_{h_8} \mapsto b\}$, respectively. It is easy to see that $\{\mathbf{C}_1, \mathbf{C}_2\}$ is a sound and complete solution set for $\mathbf{C}_{ex}$. Since $\mathbf{C}_{ex}$ is valid, such a set can be computed by the constraint solver (Fact 1).

5.3 Check the Induced Substitutions

Let $\mathbf{B}^s = \pi^s, \mathcal{T}^s_1, \ldots, \mathcal{T}^s_l$ be the symbolic branching structure obtained in the first step of SolveStrategy and let $\mathbf{C}'$ be the simple constraint system returned by the constraint solver when applied to $\mathbf{C} = \mathbf{C}_{\mathbf{B}^s}$ in the second step of SolveStrategy. Let ν be the complete solution associated with $\mathbf{C}'$ (see Section 5.2). We emphasize that for our algorithm to work, it is important that ν replaces the variables in $\mathbf{C}'$ by *new* intruder atoms from $\mathcal{A}_I$ not occurring in $\mathbf{B}^s$.

Basically, we want to check that when applying ν to $\mathbf{B}^s$, which yields $\mathbf{B}^s\nu = \pi^s\nu, \mathcal{T}^s_1\nu, \ldots, \mathcal{T}^s_l\nu$, we obtain a solution of the problem instance $(P, (C_1, C'_1), \ldots, (C_l, C'_l))$. Hence, we need to check whether i) $\pi^s\nu$ corresponds to a path in $\mathcal{G}_P$ from the initial state of $\mathcal{G}_P$ to a state $q \in \mathcal{G}_P$ and ii) $\mathcal{T}^s_i\nu$ corresponds to a q-strategy tree for (C_i, C'_i) for every i. However, since ν is a complete solution of $\mathbf{C}$, some of these conditions are satisfied by construction. In particular, $\pi^s\nu$ is guaranteed to be a path in $\mathcal{G}_P$ starting from the initial state. Also, the conditions 1.–3. of strategy trees (Definition 1) do not need to be checked and we know that $\mathcal{T}^s_i\nu$ satisfies $(C_i, \emptyset)$. Hence, SolveStrategy only needs to make sure that 4. and 5. of Definition 1 are satisfied for every $\mathcal{T}^s_i\nu$ and that $\mathcal{T}^s_i\nu$ fulfills $(\emptyset, C'_i)$. Using

that the derivation problem is decidable in polynomial time [6] (given a message m and a finite set of messages $\mathcal{I}$, decide whether $m \in d(\mathcal{I})$), all of these remaining conditions can easily be checked (see [10] for details).

In our example, the induced substitution for $\mathbf{C}_i$ is ν_i as $\mathbf{C}_i$ does not contain any variables. It can easily be verified that with $\mathbf{C}' = \mathbf{C}_2$ and the induced substitution ν_2, the above checks are all successful. However, they fail for $\mathbf{C}' = \mathbf{C}_1$ and ν_1 because in h_4 the rule $a \Rightarrow c_1$ could also be applied but it is not present in $\mathbf{B}^s_{ex}$. This violates Definition 1, 5. In fact, $\mathbf{B}^s_{ex}\nu_1$ would not yield a solution of the instance $(P_{ex}, ((C_{ex}, C'_{ex})))$. This example illustrates that in SolveStrategy one cannot dispense with the last step, namely checking the substitutions, and that one has to try the different constraint systems in the sound and complete solution set for $\mathbf{C}$.

References

[1] R.M. Amadio, D. Lugiez, and V. Vanackere. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290(1):695–740, 2002.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *CAV 2002*, LNCS 2404, pages 349–353. Springer, 2002.

[3] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Security&Privacy 2002*, pages 86–99, 1998.

[4] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In *ESORICS 2003*, LNCS 2808, pages 253–270. Springer, 2003.

[5] R. Chadha, M.I. Kanovich, and A.Scedrov. Inductive methods and contract-signing protocols. In *CCS 2001*, pages 176–185. ACM Press, 2001.

[6] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP Decision Procedure for Protocol Insecurity with XOR. In *LICS 2003*, pages 261–270. IEEE, Computer Society Press, 2003.

[7] Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols. In *ASE 2001*, pages 373–376. IEEE CS Press, 2001.

[8] P. H. Drielsma and S. Mödersheim. The ASW Protocol Revisited: A Unified View. In *ARSPA*, 2004.

[9] J.A. Garay, M. Jakobsson, and P. MacKenzie. Abuse-free optimistic contract signing. In *CRYPTO'99*, LNCS 1666, pages 449–466. Springer-Verlag, 1999.

[10] D. Kähler and R. Küsters. A Constraint-Based Algorithm for Contract-Signing Protocols. Technical report, IFI 0503, CAU Kiel, Germany, 2005. Available from http://www.informatik.uni-kiel.de/reports/2005/0503.html

[11] D. Kähler, R. Küsters, and Th. Wilke. Deciding Properties of Contract-Signing Protocols. In *STACS 2005*, LNCS 3404, pages 158–169. Springer, 2005.

[12] S. Kremer and J.-F. Raskin. Game analysis of abuse-free contract signing. In *CSFW 2002*, pages 206–220. IEEE Computer Society, 2002.

[13] J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *CCS 2001*, pages 166–175. ACM Press, 2001.

[14] M. Rusinowitch and M. Turuani. Protocol insecurity with a finite number of sessions, composed keys is NP-complete. *Theoretical Computer Science*, 299(1–3):451–475, 2003.

[15] V. Shmatikov and J.C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, 2002.

Logical Omniscience in the Semantics of BAN Logic*

Mika Cohen Mads Dam

School of Information and Communication Technology, KTH, Stockholm, Sweden
{mikac,mfd}@imit.kth.se

Abstract

BAN logic is an epistemic logic for verification of cryptographic protocols. A number of semantics have been proposed for BAN logic, but none of them capture the intended meaning of the epistemic modality in a satisfactory way. This is due to the so-called *logical omniscience problem*: Agents are "ideal reasoners" in existing semantics, while agents in BAN logic have only limited cryptographic reasoning powers. Logical omniscience is unavoidable in Kripke semantics, the standard semantical framework in epistemic logic. Our proposal is to generalize the epistemic accessibility relation of Kripke semantics so that it changes not only the current execution point, but also the currently predicated message. When instantiated on message passing systems, the semantics validates BAN logic. It makes agents introspective ("self-aware") of their own knowledge and of their own actions of sending, receiving and extracting.

Keywords: BAN logic; Epistemic logic; Kripke semantics; Security protocols; Logical omniscience problem

1 Introduction

BAN logic, proposed by Burrows, Abadi and Needham in the late eighties, is an epistemic logic for verification of cryptographic protocols ([4]). From a practical point of view, BAN logic has turned out to be quite successful: It produces short, informative derivations that can reveal subtle protocol errors. However, despite a number of semantics proposed for BAN and BAN-like logic (cf. [1, 5, 8, 10, 11, 12, 14]), the semantics of the epistemic (knowledge) modality in BAN logic remains problematic. This is a serious problem, since it makes it unclear what a proof in BAN logic establishes, and it makes an analysis of BAN logic in semantical terms, for instance using model checking, of limited value.

The basic problem when interpreting BAN's knowledge modality is the well-known *logical omniscience problem*. As an example, under BAN's idealized treatment of cryptography it is reasonable to assume the entailment M *fresh* $\models \{M\}_k$ *fresh*. However, the entailment a *knows* M *fresh* $\models$ a *knows* $\{M\}_k$ *fresh* should not be validated since in BAN logic agent a knows M is inside $\{M\}_k$ only when a knows k. From the point of view of modal logic, the example shows the failure of the *rule of normality* that allows inference of an entailment a *knows* $F_1 \models a$ *knows* F_2 from the entailment $F_1 \models F_2$. As another example, in the context of the NSSK protocol it is reasonable to assume the entailment s *said* $n, b, k, \{k, a\}_{k_b} \models k_b$ *good for* $b \cdot s$ since the former message is only ever uttered by s when it so happens that k_b is b:s server key (and therefore is good for communication between b and s). Yet, the entailment

$$a \textit{ knows } s \textit{ said } n, b, k, \{k, a\}_{k_b} \models a \textit{ knows } k_b \textit{ good for } b \cdot s \tag{1}$$

*Work supported by the Swedish Research Council grants 621-2003-2597 and 622-2003-6108

should not be validated, since in BAN logic agent a can deduce what key $\{k, a\}_{k_b}$ is locked with only if a already knows k_b. In fact, from (1) together with BAN's message meaning rule, we would get the entailment

$$a \textit{ sees } \{\textit{from } s : n, b, k, \{k, a\}_{k_b}\}_{k_a},\ a \textit{ knows } k_a \textit{ good for } a \cdot s \models a \textit{ knows } k_b \textit{ good for } b \cdot c$$

which diverges even more strongly from the intended meaning in BAN logic.

Logical omniscience (the rule of normality) is intimately tied to the use of Kripke semantics. In this type of semantics the modality *a knows* is interpreted through an epistemic accessibility relation $\sim_a$ connecting execution points that are equivalent up to a's restricted power of observation: At execution point s, a *knows* F just in case F holds at every accessible execution point s', $s \sim_a s'$.

Since all Kripke semantics validate the rule of normality, it follows that we need to look to non-Kripkean semantics to avoid validities that are unfaithful to the intended meaning in BAN logic. We suggest a generalization of Kripke semantics that lets the jump from the current execution point to an epistemically accessible execution point affect the predicated messages. The intuition is as follows. Say an agent a views a cipher text M at the current execution point s. As in Kripke semantics we assume that a may be unsure about what execution point she is at, because s and some other execution point s' share the same history up to a's observation powers. In addition, a may be unsure about what the cipher text contains, because a has observed the same properties of M at s as she would have observed of some corresponding cipher text M' at s'. For instance, if a extracts M from the third message a received at s, then a extracts M' from the third message a received at s'; if a cannot decrypt M at s, then a cannot decrypt M' at s', and so on.

To reflect the correspondence between messages at different execution points we relativize accessibility to message renamings. We write $s \sim_a^r s'$ when renaming r carries each message M at s to a corresponding message $r(M)$ at s'. With the relativized accessibility relation, a generalization of Kripke semantics is immediate:

$$s \models a \textit{ knows } F(M) \Leftrightarrow \forall s' : \forall r : s \sim_a^r s' \Rightarrow s' \models F(r(M))\ .$$

For instance, agent a knows that M is fresh, if all corresponding messages at epistemically accessible execution points are fresh.

This semantics avoids logical omniscience, since the predicated message M might change under r as we move from s to an epistemically accessible point s'. There is, however, an interesting weakening of normality which continues to hold, namely the closure of knowledge under validities that only mention keys used by the agent.

$$F_1 \models F_2 \Rightarrow a \textit{ uses } \textit{Keys}(F_1, F_2),\ a \textit{ knows } F_1 \models a \textit{ knows } F_2$$

where $\textit{Keys}(F_1, F_2)$ contains all message terms that are applied as keys in F_1 and F_2. To illustrate, from the entailment x *fresh* $\models \{x\}_y$ *fresh* we can infer the entailment a *uses* y, a *knows* x *fresh* $\models a$ *knows* $\{x\}_y$ *fresh*. By universal substitution of message terms for variables, we can then conclude the entailment

$$a \textit{ uses } K,\ a \textit{ knows } M \textit{ fresh} \models a \textit{ knows } \{M\}_K \textit{ fresh} \tag{2}$$

for arbitrary (complex) message terms K and M, even when keys other than K are applied in M.

After instantiating the semantics on message passing systems, we show that agents are introspective of their own knowledge, i.e. the modal logic *S5* axioms hold, as is the custom in computer science applications of epistemic logic. Furthermore we show that agents are introspective of their own actions of sending, receiving and extracting (decryption and un-pairing of received messages). For instance, we show introspection of received messages: *a received M* $\models$ *a knows a received M*. While this is immediate from the truth condition for knowledge, it is rather significant. Firstly, it is the central point when validating BAN logic. The unsoundness of BAN logic in related Kripke semantics, such as [1, 12, 14], can ultimately be tied back to the fact that agents are not introspective (in

the above sense) of their received messages [1]. As soon as a Kripke semantics hides part of an agents local state to the agent herself, as these semantics do, we lose introspection of received messages. Secondly, introspection of received messages in combination with the above weakening of normality has an interesting implication: knowledge of cryptographic structure may at times transcend the discerning power of the keys used.

We complete the model construction by interpreting the atomic BAN predicates on message passing systems and show soundness of BAN logic. The interpretation we propose involves a fixed point construction to identify keys used with keys known, a construction which may be of independent interest. Finally the paper is closed by a discussion of related and future work, in particular the prospects for using the weakened rule of normality to eliminate BAN's idealization step.

Our semantical investigations so far cover only the symmetric key part of BAN logic. We expect no difficulties in extending the semantics to asymmetric cryptography.

2 BAN Logic

Language Assume a set of agents $a, b, ...$, a set of *message atoms* $k, n, ...$, a set of *message variables* $x, y, z, ...$, and a set of *atomic predicates* p. The set of *message terms* and *statements* are defined by:

$$\textit{Statements } F ::= \ p(M) \mid \textit{a knows } F$$
$$\textit{Message terms } M, M' ::= \ F \mid a \mid k \mid x \mid M, M' \mid \{M\}_{M'} \mid \textit{from a : M}$$

A closed message term, or *message*, is a message term with no variables. A message term is open if it is not closed. Though the BAN language lacks negation, we prove a result (Theorem 9.2) for a language extended with negation ($\neg$) of statements.

Intuitively, atomic statement $p(M)$ expresses the proposition that message M satisfies property p, the operator $\cdot, \cdot$ represents pairing of messages, the operator $\{\cdot\}_\cdot$ represents encryption and the operator *from* $\cdot : \cdot$ represents sender field annotation. Message terms include sender field annotations and statements, because as BAN logic is usually applied, it proves properties of so called *idealized* protocols, protocols where messages may include a sender field and messages may contain statements expressing propositions.

The set of atomic predicates includes, at least, the four *atomic BAN predicates*: *a sees*, *a said*, *fresh*, *good for* $a{\cdot}b$ as well as the special atomic predicate *a uses*. Their intended informal meaning is as follows. The predicate *a sees* is true of a message if a can extract the message from something a received. Analogously, *a said* is true of a message if a can extract the message from something a sent. A message *fresh* if it did not circulate until recently. A message satisfies *good for* $a \cdot b$ if every circulated message encrypted with this message as key was said by a or b. Finally, *a uses* a message if a uses that message as a key for decryption and encryption.

Proof rules The rules of BAN logic are summarized in Table 1. We use *Knows* to represent an arbitrary sequence of 0 or more epistemic modalities. Table 1 leaves some conditions implicit: We have omitted symmetric variations and closure under cut and weakening. Note that certain rules assume that agents do not misuse idealizations. For instance, rule *R1*, the *message meaning rule*, assumes that sender fields inside cipher texts are reliable. Also, rule *R7*, the *nonce verification rule*, assumes that agents only say statements known to be true while fresh.

While the original BAN paper ([4]) reads the epistemic modality as "agent a believes that", BAN logic is intuitively consistent with a knowledge interpretation. As in [8, 10], we adopt a knowledge interpretation and add the axiom T. The atomic BAN predicate $jurisdiction$ thereby becomes superfluous, and is therefore removed. For a more detailed discussion we refer the reader to [8]. Notice that we generalize the customary modal logic axiom T (*a knows* $F \vdash F$) to arbitrary iterations of epistemic modalities, by adding *Knows* to antecedent and consequent.

[1] Only [1] was intended to validate BAN.

$R1$. *a sees* $\{$*from b :* $M\}_{M'}$, *a knows* M' *good for* $a \cdot b \vdash a\,knows$ *b said* M
$R2$. *a knows* M *fresh* $\vdash$ *a knows* M, M' *fresh*
$R3$. *a knows* M *fresh*, *a knows* M' *good for* $a \cdot b \vdash$ *a knows* $\{M\}_{M'}$ *fresh*
$R4$. *a sees* $M, M' \vdash$ *a sees* M
$R5$. *a sees* $\{M\}_{M'}$, *a knows* M' *good for* $a \cdot b \vdash$ *a sees* M
$R6$. *a knows b said* $M, M' \vdash$ *a knows b said* M
$R7$. *a knows* $M_1, ..., F, ..., M_n$ *fresh*, *a knows b said* $M_1, ..., F, ..., M_n, \vdash$ *a knows b knows* F
T. *Knows a knows* $F \vdash$ *Knows* F

Table 1: BAN proof rules

3 Semantics for the Non-Epistemic Language Fragment

In computer science, epistemic logics are customarily interpreted on *multi-agent systems* [6], pairs $\mathsf{S} = \langle S, | \rangle$, where S is a non-empty set of execution points and $|$ is a local state projection assigning a local state $s|a$ to each agent a and execution point s. Intuitively, the local state contains all the data currently accessible to that agent. For instance, when modeling a communication protocol, the local state of an agent might be derived from the initial condition plus the sequence of send and receive actions she has performed so far. A *multi-agent model* on S is a triple $\mathsf{M} = \langle S, |, I \rangle$, where I is an interpretation of atomic predicates. That is, to each atomic predicate p and each execution point $s \in S$, the interpretation I assigns the set $I(p, s)$ of messages (closed message terms) that satisfy p at s.

Closed statements are true w.r.t. an execution point s in a model M. The truth condition for atomic closed statements and negation (of closed statements) are as expected: $s \models_{\mathsf{M}} p(M) \Leftrightarrow M \in I(p, s)$ and $s \models_{\mathsf{M}} \neg F \Leftrightarrow s \not\models_{\mathsf{M}} F$. The truth condition for epistemic closed statements is left to section 4. Open statements are true w.r.t. an assignment V of messages to message variables, and an execution point s in a model M. Assignments are lifted to arbitrary message terms in the usual way; write $|M|_V$ for the value of M under V. The truth condition for open statements is: $V, s \models_{\mathsf{M}} F(M) \Leftrightarrow s \models_{\mathsf{M}} F(|M|_V)$.

If Δ is a set of statements, we write $V, s \models_{\mathsf{M}} \Delta$ if $V, s \models_{\mathsf{M}} F$, for all $F \in \Delta$. If C is a class of models: $\Delta \models_{\mathsf{C}} F$, if and only if, for all models M in C, for all execution points s in M and for all assignments V, if $V, s \models_{\mathsf{M}} \Delta$ then $V, s \models_{\mathsf{M}} F$.

4 Semantics for Knowledge

We interpret the epistemic modality through a generalized accessibility relation $\sim_a$ that relates not only execution points, but also messages at one execution point to messages at another. The intuition is that a cipher text M at the current execution point s may correspond, for a, to a different cipher text M' at an epistemically accessible execution point s'. That is, M at s could, for all a knows, be M' at s'. Let r be a *renaming* of messages, i.e. a function in the set of messages, defined for all messages. If r maps every message at s to a corresponding message at s', we say that r is a *counterpart mapping* between s and s' for agent a, and write $s \sim_a^r s'$. Given this ternary accessibility relation $\sim_a$, Kripke semantics can be generalized in an obvious way:

$$s \models_{\mathsf{M}} \textit{a knows F(M)} \Leftrightarrow \forall s' \in S : \forall r : s \sim_a^r s' \Rightarrow s' \models_{\mathsf{M}} F(r(M)) \; .$$

Here, $F(M)$ is any statement in the message term M. We do not assume that message M is somehow accessible to agent a in s, such as once said, or seen, by a. Agents may well know things about messages that are not accessible to them. In fact, this is an essential part of BAN logic (as witnessed by, for instance, axiom *R2*).

Counterpart mappings must be transparent to the set of available keys. A renaming r is *transparent* to a set Π of messages, in symbols $\Pi \rhd r$, if r respects all cryptographic structure accessible when using Π as keys: Π (used

C1. $M' \in \Pi \Rightarrow r(\{M\}_{M'}) = \{r(M)\}_{r(M')}$	C2. $r(M, M') = r(M), r(M')$
C3. r is injective	C4. r is surjective
C5. $r(F(M)) = F(r(M))$	C6. $r($*from a : M*$) =$ *from r(a) : r(M)*
C7. $r(k) = k$, k is agent name or message atom	

Table 2: Requirements for $\Pi \rhd r$

as keys) cannot distinguish a sequence $M_1, M_2, ...$ from $r(M_1), r(M_2), ...$ Formally, we stipulate that $\Pi \rhd r$, if and only if, each condition in Table 2 above is satisfied. Condition C1 says that encryption structure is plain, or clear, when the appropriate key is available, condition C2 says that pairing structure is always plain, conditions C3 and C4 say that distinct messages appear distinct, condition C5 says that atomic predicates and propositional operators are plain text, condition C6 says that sender field structure is plain, and condition C7, finally, says that agent names and message atoms are plain text.

Lemma 4.1

1. $\Pi \rhd \iota$ *where* ι *is the identity on messages*
2. $\Pi \rhd r, \;\; r(\Pi) \rhd r' \Rightarrow \Pi \rhd (r' \circ r)$
3. $\Pi \rhd r \Rightarrow r(\Pi) \rhd r^{-1}$
4. $\Pi \rhd r, \;\; \Pi \supseteq \Pi' \Rightarrow \Pi' \rhd r$

Proof. (1) and (4) are immediate. We prove (2) here. The proof of (3) is similar. Assume $\Pi \rhd r$ and $r(\Pi) \rhd r'$. Only requirement C1 of Table 2 is non-trivial. Assume $M' \in \Pi$. By the assumptions, $r(\{M\}_{M'}) = \{r(M)\}_{r(M')}$ and $r(M') \in r(\Pi)$. Thus, $r'(\{r(M)\}_{r(M')}) = \{r'(r(M))\}_{r'(r(M'))} = \{(r' \circ r)(M)\}_{(r' \circ r)(M')}$, i.e., $(r' \circ r)(\{M\}_{M'}) = r'(\{r(M)\}_{r(M')}) = \{r'(r(M))\}_{r'(r(M'))} = \{(r' \circ r)(M)\}_{(r' \circ r)(M')}$. □

Counterpart mappings must, furthermore, respect the current local state of the agent; we assume a renaming can be lifted pointwise to a permutation on local states. For r to be a counterpart mapping between s and some other point s', we require that r transforms the local state of the agent at s into her local state at s'.

The idea, then, is to relate the states s and s' under the renaming r for agent a, in symbols $s \sim_a^r s'$, just in case r transforms the local state of a at s into the local state of a at s' and r respects the keys used by the agent at s:

$$s \sim_a^r s' \Leftrightarrow r(s|a) = s'|a \text{ and } I(a \textit{ uses}, s) \rhd r \; . \tag{3}$$

Each multi-agent model thus determines a unique ternary epistemic accessibility relation $\sim_a$. In section 9 below we address the apparent asymmetry of (3) and show that under the definitions of *uses* which we consider, whenever $s \sim_a^r s'$ then $s' \sim_a^{r^{-1}} s$.

5 Crypto Normality

The semantics avoids logical omniscience (the rule of normality). To see this, let $S = \{s\}$, $I(p, s) = \{\{M\}_{M'}\}$, $I(a \textit{ uses}, s) = \emptyset$ and $s|a = \emptyset$. Then there is a renaming r such that $r(\{M\}_{M'}) \neq \{M\}_{M'}$ and $s \sim_a^r s$. Thus, $\not\models_{\mathsf{M}}$ *a knows* $p(\{M\}_{M'})$. Yet, $\models_{\mathsf{M}} p(\{M\}_{M'})$.

There is, however, an interesting weakening of normality which continues to hold. To formulate this, let *Keys*(M) be the set of message terms applied as keys in M such that *Keys*$(\{M\}_{M'}) =$ *Keys*$(\{M'\} \cup M) \cup$ *Keys*(M'), *Keys*$(M, M') =$ *Keys*$(M) \cup$ *Keys*(M'), *Keys*$($*from a : M*$) =$ *Keys*(M), *Keys*$(P(M)) =$ *Keys*(M), *Keys*$(k) = \emptyset$, if k is message atom or agent name, and *Keys*$(x) = \emptyset$, for message variables x. For example,

Keys$(\{w, \{x, k\}_y\}_z) = \{y, z\}$. Let *Keys*$(\Pi) = \cup_{M \in \Pi}$*Keys*$(M)$, write *a uses* Π for the set $\{$*a uses M* $\mid M \in \Pi\}$, and write *a knows* Δ for the set $\{$*a knows* $F \mid F \in \Delta\}$.

Lemma 5.1 $|\text{Keys}(M)|_V \rhd r \Rightarrow r(|M|_V) = |M|_{r \circ V}$

Proof. By induction over the structure of M. The base step, where M is a variable, an agent name or message atom, is immediate from requirement C7 of Table 2. For the induction step, assume that the property holds for messages M_1 and M_2, i.e. $|$*Keys*$(M_1)|_V \rhd r \Rightarrow |M_1|_{r \circ V} = r(|M_1|_V)$ and $|$*Keys*$(M_2)|_V \rhd r \Rightarrow |M_2|_{r \circ V} = r(|M_2|_V)$. Assume $|$*Keys*$(\{M_1\}_{M_2})|_V \rhd r$. Then, $|$*Keys*$(M_1)|_V \cup |Keys(M_2)|_V \cup \{|M_2|_V\} \rhd r$. By the induction assumption and Lemma 4.1.4, $|M_1|_{r \circ V} = r(|M_1|_V)$ and $|M_2|_{r \circ V} = r(|M_2|_V)$. Then, by requirement C1 of Table 2, $r(|\{M_1\}_{M_2}|_V) = r(\{|M_1|_V\}_{|M_2|_V}) = \{r(|M_1|_V)\}_{r(|M_2|_V)} = \{|M_1|_{r \circ V}\}_{|M_2|_{r \circ V}} = |\{M_1\}_{M_2}|_{r \circ V}$. Showing that pairing and idealization constructions preserve the property is analogous. □

From Lemma 5.1 we get the weak normality rule.

Theorem 5.2 (Crypto Normality) *If* $\Delta \models_M F$ *then a uses Keys*(Δ, F), *a knows* $\Delta \models_M$ *a knows* F.

Crypto normality says that an agents knowledge is closed under logical validities in which all the keys applied are used by the agent. By itself, crypto normality may appear overly restricted, since all keys used in Δ or F must also be used by a. Crypto normality becomes more powerful, however, when combined with the rule of substitution.

Theorem 5.3 (Rule of Substitution) *Let* σ *be any substition of (possibly open) message terms for message variables. If* $\Delta \models_M F$ *then* $\sigma(\Delta) \models_M \sigma(F)$.

In conjunction, the two rules allow interesting inferences to be made, such as (2) in section 1.

6 Message Passing Systems

We instantiate models in message passing systems (cf. [6]), as in the BAN literature. Since the definitions are standard and well-known, we will only briefly hint at them. In a message passing system execution proceeds in rounds. During the first round, initial shared and private possessions are established. From then on, at each round, every agent either sends a message, receives a message or performs some unspecified internal action. By a *message passing model* we mean a multi-agent system $\mathsf{M} = \langle S, |, I \rangle$ based on a message passing system S. We require that the local state $s|a$ of an agent a consists of a first round of initializations followed by a's local history of send and receive actions. As an immediate consequence, agents know which messages they send and receive. Assume predicates *a received* and *a sent*, with *I(a received, s)* = $\{M \mid a$ has received *M* at $s\}$, and *I(a sent, s)* interpreted analogously. The following introspection principle is easily seen to be valid:

Proposition 6.1 (Receive and send introspection) *For message passing models:*

1. *a received* $M \models$ *a knows a received* M
2. *a sent* $M \models$ *a knows a sent* M

To see this, assume $s \sim_a^r s'$. Then, $r(s|a) = s'|a$, i.e. if a received M at s then a received $r(M)$ at s', and correspondingly for messages sent by a. While easily proved, Proposition 6.1 is nonetheless of some consequence. To begin with, the unsoundness of BAN logic in related Kripke semantics, such as [1, 12, 14], ultimately ties back to the failure of Proposition 6.1. When a Kripke semantics hides part of an agents local state from the agent, as these semantics do, we lose receiving and sending introspection: Say a received a cipher text M at s. Then there might be some point s' which is indistinguishable for a from the current point s, but where a received a different cipher text M', not M. Moreover, Proposition 6.1 in combination with crypto normality (Theorem 5.2) has some interesting, and perhaps surprising, implications for knowledge of cryptographic structure. We explore these implications in the section 7.

7 Knowledge of the Unseen

Prima facie it might be thought that an agents knowledge of cryptographic structure depends solely on what keys she uses. However, the mere finding of a cipher text at a certain place might alone indicate something about its contents. For instance, after the second protocol step in the Needham Shröder shared key protocol (NSSK) between principals a and b and with key server s, agent a knows the contents of the ticket she is to forward to b, despite the fact that she cannot decrypt it. The semantics respects such intuitions. To illustrate, assume that message passing model M implements NSSK between a, b and s. We may expect the following: *a received* $\{n, b, k, x\}_{k_a}$, k_a *good for* $a \cdot s \models_{\mathsf{M}} x$ *contains* k, a. (The meaning of *contains* should be clear from the context, while the precise semantics of *good* is not an issue in this example.) By crypto normality (Theorem 5.2) and universal substitution (Theorem 5.3), *a knows a received* $\{n, b, k, \{k, a\}_{k_b}\}_{k_a}$, *a knows* k_a *good for* $a \cdot s$, *a uses* $k_a \models_{\mathsf{M}}$ *a knows* $\{k, a\}_{k_b}$ *contains* k, a. By receiving introspection (Proposition 6.1), *a received* $\{n, b, k, \{k, a\}_{k_b}\}_{k_a}$, *a knows* k_a *good for* $a \cdot s$, *a uses* $k_a \models_{\mathsf{M}}$ *a knows* $\{k, a\}_{k_b}$ *contains* k, a. Thus, if k_a is a's server key and a receives $\{n, b, k, \{k, a\}_{k_b}\}_{k_a}$, then a knows the contents of $\{k, a\}_{k_b}$ even though a is not using k_b as a key.

The reason why the semantics supports deductions such as the above is that the set of counterpart mappings is limited not only by the current keys, but also by the current local state. Say renaming r is transparent to the keys used at the current point s, in symbols $I(a \textit{ uses}, s) \rhd r$. This does not guarantee, however, that r is a counterpart mapping from s to any execution point s': There might be no s' in the given system such that $r(s|a) = s'|a$. In this case the agent can rule out r even though r is transparent to her current keys.

8 Interpreting BAN's Atomic Predicates

To complete the semantics for BAN logic, only the atomic predicates remain. This is a subject of subtle and somewhat bewildering variability (cf. [1, 8, 10]). We do not claim our definitions are canonical. Our goal is to show that the renaming semantics can be completed to a meaningful interpretation which validates BAN.

The way the predicates are explained informally in section 2, once the interpretation of *uses* is fixed, the interpretation of *sees*, *said* and *good* follow in a fairly straightforward fashion. Specifically, for *sees* we require that $I(a \textit{ sees}, s)$ is the smallest set Π that includes a's initial possessions, the messages a has received at s and such that Π is closed under decryption with keys used ($\{M\}_{M'} \in \Pi$ and $M' \in I(a \textit{ uses}, s) \Rightarrow M \in \Pi$) and un-pairing ($M, M' \in \Pi \Rightarrow M \in \Pi$ and $M' \in \Pi$) and sender-field removal (*from b:* $M \in \Pi \Rightarrow M \in \Pi$). The predicate *said* is defined analogously for sent messages, except that a's initial possessions are not included. For *good* we require that $M \in I(\textit{good for } a \cdot b, s)$, if and only if, whenever $\{M'\}_M$ is a sub term of some message in $I(c \textit{ received}, s)$, then both M' and $\{M'\}_M$ are in $I(a \textit{ said}, s)$ or both M' and $\{M'\}_M$ are in $I(b \textit{ said}, s)$, for any agent c and any message M'. We leave the interpretation of the predicate *fresh* open, merely requiring that it is independent of the interpretation of *uses* and that it is closed under the sub-term relation ($M \in I(\textit{fresh}, s) \Rightarrow M, M' \in I(\textit{fresh}, s)$, $M', M \in I(\textit{fresh}, s)$, $\{M\}_{M'} \in I(\textit{fresh}, s)$, and $\{M'\}_M \in I(\textit{fresh}, s)$). One could satisfy these requirements by defining, similarly to [8], a message as fresh if it is not a subterm of any message said by anyone more than d rounds back, for some fixed d. Interpreting the predicate *fresh* is somewhat problematic, but it is peripheral to the issues addressed in this paper. We refer the reader to [8, 10] for more detailed discussions.

We then turn to the predicate *uses*. An immediate observation is that the interpretation of *uses* must validate the entailment

$$\text{a } \textit{knows } M \textit{ good for } a \cdot b \models a \textit{ uses } M \ . \tag{4}$$

This requirement is fundamental, since otherwise rules *R1*, *R3*, and *R5* (Table 1) will not be validated.

A possible approach to the definition of *uses* is to view *uses* and *sees* as synonyms, so that a key is used by an agent just in case it is possessed initially or it is received, or it can be obtained by decryption and un-pairing from used messages. This kind of "operational" view is taken, with variations, in most papers on se-

mantics for BAN like logics. The problem with this definition is that it does not validate (4), unless the class of message passing systems is restricted in some way. For instance a model M may satisfy an entailment such as: a *receives* $k, a, b \models_{\mathsf{M}} k, x$ *good for* $a \cdot b$. Then, by crypto normality (Theorem 5.2) and receive introspection (prop. 6.1.1), a *receives* $k, a, b \models_{\mathsf{M}} a$ *knows* k, x *good for* $a \cdot b$, but it might well be that a has not seen k, x, contradicting (4). This counterexample can be fixed, of course, by disallowing complex terms as keys. But other, similar counterexamples would still require restricting the class of allowed message passing systems. For instance, if we allowed a model specific dependency between properties of different message atoms, say a *receives* $k, a, b \models_{\mathsf{M}} k'$ *good for* $a \cdot b$, then a might be able to conclude that k' is good without actually seeing it, again contradicting (4).

We propose an alternative definition of *uses* which we believe is of independent interest. The idea is to consider a key to be used by an agent just in case the agent knows some property p of that key. Since properties (sees, said, etc.) are defined by means of *uses* itself, a recursive definition is called for. An inductive, rather than a coinductive, definition seems appropriate, since *a uses* should contain the set of keys that a has gathered some positive information about. Adopting this approach we thus define the interpretation function on a message passing system S as a least interpretation function I (in an order extended point wise from set containment) such that $s \models_{\langle S,I \rangle}$ *a uses M*, if and only if, $s \models_{\langle S,I \rangle}$ *knows* $p(M)$ for some atomic BAN predicate p. (We leave the local state projection $|$ implicit.) If we call models that use this definition of the interpretation function *inductive*, we obtain:

Theorem 8.1 *Every message passing system determines a unique inductive model.*

Proof. Assume a message passing system S. The interpretation function in an inductive model on S is, by definition, the least fixed point of the following function f that assigns an interpretation function $f(I)$ to every possible interpretation function I on S. For predicate *uses*, $f(I)(a \textit{ uses}, s) = \{M \mid \exists \text{ atomic BAN predicate } p : s \models_{\langle S,I \rangle} a \textit{ knows } p(M)\}$ and, for atomic BAN predicates p, $f(I)(p, s)$ is defined with $f(I)(a \textit{ uses}, s')$ as the keys used for any agent a at any point s'. From lemma 4.1.4, f is monotone. Therefore, f has a least fixed point. □

Inductive models obviously satisfy the requirement (4) above. In fact, as far as requirement (4) is concerned, we could have defined *uses* in terms of predicates *good* alone, so that $s \models a$ *uses M*, if and only if, $s \models a$ *knows M good for* $a \cdot b$ for some agent b. Perhaps, such a solution would be even more faithful to intuitions in BAN logic, but it would not be quite satisfactory for some protocols (Yahalom is an example) where keys need to be used before they are known to be good. Inductive models offer, in our opinion, an interesting, more extensional, alternative to the more traditional operational models.

9 Introspection Properties

We have already seen (Proposition 6.1) that agents in message passing models are introspective of their received and sent messages. In this section, we observe some further introspection properties in inductive models. We emphasize that these results also hold for models based on an operational interpretation of *uses*.

Lemma 9.1 *For inductive models M:*

1. $s \sim_a^{\iota} s$
2. $s \sim_a^{r} s',\ \ s' \sim_a^{r'} s'' \Rightarrow s \sim_a^{r' \circ r} s''$
3. $s \sim_a^{r} s' \Rightarrow s' \sim_a^{r^{-1}} s$

Proof. (1) Immediate from Lemma 4.1.1. The proof of (2) is similar to (3) and left out. For (3) we first prove, using fixed point induction, that $s \sim_a^r s' \Rightarrow I(a\ \mathit{uses}, s') \subseteq r(I(a\ \mathit{uses}, s))$ where I is the interpretation function in M. Let I_j be the interpretation function at step j in the fixed point construction of the proof of Theorem 8.1, such that $I_0 = \emptyset$, $I_{j+1} = f(I_j)$, and $I_\delta = \cup_{j<\delta} I_j$, if δ is a limit ordinal. Let M_{I_j} be M with the interpretation I replaced by I_j. We show for all j that

$$I_j(a\ \mathit{uses}, s) \rhd r \wedge r(s|a) = s'|a \Rightarrow I_j(a\ \mathit{uses}, s') \subseteq r(I_j(a\ \mathit{uses}, s)) \tag{5}$$

The property holds for I_0, since $I_0(a\ \mathit{uses}, s') = \emptyset$. For successor ordinals, assume (5) holds for j. Assume $I_{j+1}(a\ \mathit{uses}, s) \rhd r$ and $r(s|a) = s'|a$. Pick any message M' such that $M' \in I_{j+1}(a\ \mathit{uses}, s')$. By C4 in Table 2, $M' = r(M)$ for some message M. Then $s' \models_{\mathsf{M}_{I_{j+1}}} a\ \mathit{uses}\ (r(M))$. By the definition of I_{j+1}, there is an atomic predicate p such that $s' \models_{\mathsf{M}_{I_j}} a\ \mathit{knows}\ p(r(M))$. Since $I_j \subseteq I_{j+1}$, by Lemma 4.1.4, $I_j(a\ \mathit{uses}, s) \rhd r$. By Lemma 4.1.3, $r(I_j(a\ \mathit{uses}, s)) \rhd r^{-1}$, so by the induction hypothesis and Lemma 4.1.4, $I_j(a\ \mathit{uses}, s') \rhd r^{-1}$. Since $M' = r(M)$, we want to show that $M \in I_{j+1}(a\ \mathit{uses}, s)$. By definition of I_{j+1}, it suffices to show that $s \models_{\mathsf{M}_{I_j}} a\ \mathit{knows}\ p(M)$. So pick any renaming r' and any execution point $s'' \in S$ such that $I_j(a\ \mathit{uses}, s) \rhd r'$ and $r'(s|a) = s''|a$. Since $I_j(a\ \mathit{uses}, s) \rhd r$, by the induction hypothesis, and conditions C3 and C4 of Table 2, $r^{-1}(I_j(a\ \mathit{uses}, s')) \subseteq I_j(a\ \mathit{uses}, s)$. By Lemma 4.1.4, $r^{-1}(I_j(a\ \mathit{uses}, s')) \rhd r'$. By Lemma 4.1.2 it follows that $I_j(a\ \mathit{uses}, s') \rhd r' \circ r^{-1}$. By the assumptions on r' we get that $r' \circ r^{-1}(s'|a) = r'(r^{-1}(s'|a)) = r'(s|a) = s''|a$. Since we showed $s' \models_{\mathsf{M}_{I_j}} a\ \mathit{knows}\ p(r(M))$ we obtain that $s'' \models_{\mathsf{M}_{I_j}} p(r' \circ r^{-1} \circ r(M))$. Since r' and s'' are arbitrary, it follows that $s \models_{\mathsf{M}_{I_j}} a\ \mathit{knows}\ p(M)$ which completes the successor part of the induction argument. The limit case is routine.

For the proof of the main statement (3), assume then that $s \sim_a^r s'$, i.e. $I(a\ \mathit{uses}, s) \rhd r$ and $r(s|a) = s'|a$. By Lemma 4.1.3, $r(I(a\ \mathit{uses}, s)) \rhd r^{-1}$. We also obtain, from the above induction, that $I(a\ \mathit{uses}, s') \subseteq r(I(a\ \mathit{uses}, s))$. By Lemma 4.1.4, $I(a\ \mathit{uses}, s') \rhd r^{-1}$, so $s' \sim_a^{r^{-1}} s$, which completes the proof. □

Using Lemma 9.1 the modal logic S5 properties follow directly.

Theorem 9.2 (Knowledge introspection) *For inductive models:*

1. *a knows* $F \models F$
2. *a knows* $F \models$ *a knows a knows* F
3. $\neg$ *a knows* $F \models$ *a knows* $\neg$ *a knows* F

Validity (1) in Theorem 9.2 is, of course, not an introspection property. Rather, it can be seen as the distinguishing line between knowledge and belief. In fact, (1) holds in all models, not only inductive models. From Theorem 9.2, it follows that agents are also introspective of used and seen messages:

Corollary 9.3 (Use and sees introspection) *For inductive models:*

1. *a uses* $M \models$ *a knows a uses* M
2. *a sees* $M \models$ *a knows a sees* M

Proof. (1) is immediate from Theorem 9.2. (2) follows from crypto normality (Theorem 5.2), rule of substitution (Theorem 5.3), receive introspection (Proposition 6.1.1), and use introspection (1). □

Loosely speaking, sees introspection implies that agents are introspective of extracted messages. Since sees introspection depends on receive introspection (Proposition 6.1) it fails in the related Kripke semantics of [1, 12, 14]. For similar reasons (see section 6), use introspection also fails in these semantics, when cipher texts are allowed as keys.

10 Soundness of BAN logic

As observed in section 2, some BAN rules assume that agents do not misuse idealizations. Accordingly, in our soundness result we restrict attention to *honest* models, models where *from b:* $M \in I(a\ said\ , s) \Rightarrow a = b$ and where $M_1, ..., F, ..., M_n$ *fresh* , *a said* $M_1, ..., F, ..., M_n \models$ *a knows F*. Again, we refer the reader to [8] for details.

Soundness for each BAN rule (Table 1) is now a rather immediate application of the following corollary, where *a knows* $\{M_1, ..., M_n\}$ *good* is short for *a knows* M_1 *good for* $a \cdot b_1$, *..., a knows* M_n *good for* $a \cdot b_n$.

Corollary 10.1 *Let* σ *be any substitution of message terms for variables. For inductive models M: If* $\Delta \models_M F$ *then a knows* $\sigma(Keys(\Delta, F))$ *good*, *a knows* $\sigma(\Delta) \models_M$ *a knows* $\sigma(F)$.

Proof. Immediate from crypto normality (Theorem 5.2), rule of substitution (Theorem 5.3) and requirement (4) in section 8. □

Theorem 10.2 *BAN logic is sound w.r.t. honest inductive models.*

Proof. Rule R4 (Table 1) is immediate. Rule R5 is immediate from requirement (4) in section 8. Each remaining rule is a direct application of Corollary 10.1 on some trivial validity. For instance, rule R3 follows from the fact that y *fresh* $\models \{y\}_z$ *fresh*. Rule R1 needs, in addition, sees introspection (Corollary 9.3.2), while rule *T* needs Theorem 9.2.1. □

11 Related Work

Our use of a ternary accessibility relation is most closely related to possibility relations in counterpart semantics [9]. It is, as far as we know, the first computationally grounded such semantics in epistemic logic.

In the BAN logic literature the semantics most closely related to ours are the Kripke semantics of [1, 12, 14] where the local state of an agent is partly hidden from the agent. In our framework we can recover a binary accessibility relation similar to those used in [1, 12, 14] by letting $s \sim_a s'$ iff $s \sim_a^r s'$ for some renaming r. In fact, our notion of transparent renaming can be seen as related to the message congruences of [1], and to the states of knowledge and belief of [3, 13]. As we have pointed out, however, a Kripke semantics resulting from such a binary accessibility relation $\sim_a$ is both too strong and too weak for BAN: It makes agents logically omniscient, yet fails essential introspection principles [2].

There are, of course, semantics in the literature that do in fact avoid logical omniscience (cf. [6]). But no such semantics has been shown to work for BAN-like logics. Furthermore, these semantics tend to break rather more radically than ours with Kripke semantics. One possible approach is to subdivide knowledge into an implicit and an explicit part. Implicit knowledge would be "ideal" knowledge to which logical omniscience applies, and explicit knowledge would be somehow circumscribed to reflect agents limited reasoning abilities. For instance, [7] specifies adversary capabilities in terms of abstract knowledge extraction algorithms, and [2] uses an awareness predicate to constrain, at each state, the predicates which of which an agent is aware, related to the comprehended messages of [12].

12 Conclusion

We have introduced a semantics that validates BAN logic, yet avoids the rule of normality (logical omniscience). The semantics satisfies crypto normality, a weak version of normality that filters out infeasible cryptographic reasoning powers. The semantics makes agents introspective of their own knowledge and their own actions of

[2] But we acknowledge that only [1] was intended as a semantics for BAN.

sending, receiving and extracting. We have showed how knowledge of cryptographic structure may at times transcend the discriminatory power of the keys used. Finally, we found that knowledge and keys used could be defined as simultaneous fixed points, making the keys used equal to the keys known.

A semantical foundation for BAN logic opens up the possibility of sound model checking of BAN logic specifications. Also, the semantics might be used to improve various elements of the protocol verification process in BAN. The crypto normality rule is a case in point. Using this rule we can sidestep the often criticized "idealization step" in BAN verifications. To illustrate, say we want to establish the following property of NSSK:

$$a \textit{ knows } k_a \textit{ good for } a \cdot s,\ a \textit{ knows n fresh},\ a \textit{ sees } \{n, b, k, \{k, a\}_{k_b}\}_{k_a} \models a \textit{ knows } k \textit{ good for } a \cdot b \tag{6}$$

As BAN is usually applied, one would instead prove a property of an "idealization" of the protocol where the message $\{n, b, k, \{k, a\}_{k_b}\}_{k_a}$ has been annotated with sender field and the goodness predicate. As an alternative, we introduce non-epistemic protocol specific validities:

$$k_a \textit{ good for } a \cdot s,\ \textit{n fresh},\ s \textit{ said } \{n, b, k, x\}_{k_a} \models k \textit{ good for } a \cdot b \tag{7}$$

$$k_a \textit{ good for } a \cdot s \models \neg\ a \textit{ said } \{n, b, k, x\}_{k_a} \tag{8}$$

which arguably express the required properties of the protocol rather more precisely. Starting from a (protocol independent) triviality,

$$\neg\ a \textit{ said } \{x\}_y,\ a \textit{ sees } \{x\}_y,\ y \textit{ good for } a \cdot s \models s \textit{ said } \{x\}_y, \tag{9}$$

we get specification (6) by lifting (7), (8) and (9) to epistemic validities using crypto normality (Corollary 10.1), then applying sees introspection (Corollary 9.3) and knowledge introspection (Theorem 9.2)

We have focused on BAN logic, not in particular deference to BAN, but simply because BAN is the standard logic in its family. A first question to answer is whether our semantics really captures the intended meaning of BAN formulas. A completeness result for a collection of rules which stays acceptably close to BAN's original set-up would help answer this question affirmatively, and we are currently working to address this issue.

It would be of interest also to use our semantics to support epistemic security protocol logics beyond the propositional level. An extension to first-order μ-calculus with rudimentary temporal operators would allow the BAN primitives to be defined, and thus eliminate much of the apparent arbitrariness in the choice of basic vocabulary in the BAN literature. Furthermore, a first-order extension would allow reasoning that exploits partial knowledge of complex data structures; this may be useful in the context of e.g. payment protocols, where different parts of the negotiated data structure remain hidden from different principals.

References

[1] Martín Abadi and Mark Tuttle. A semantics for a logic of authentication. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 201–216. ACM Press, August 1991.

[2] Rafael Accorsi, David A. Basin, and Luca Viganò. Towards an awareness-based semantics for security protocol analysis. *Electr. Notes Theor. Comput. Sci.*, 55(1), 2001.

[3] Pierre Bieber. A logic of communication in hostile environments. In *Proceedings of the Computer Security Foundation Workshop III*, pages 14–22. IEEE Computer Society Press, 1990.

[4] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[5] Anthony H. Dekker. C3po: A tool for automatic sound cryptographic protocol analysis. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*, pages 77–87. IEEE Computer Society Press, 2000.

[6] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[7] Joseph Y. Halpern and Riccardo Pucella. Modeling adversaries in a logic for security protocol analysis. In *FASec*, pages 115–132, 2002.

[8] Joseph Y. Halpern, Riccardo Pucella, and Ron van der Meyden. Revisiting the foundations of authentication logics. Manuscript, 2003.

[9] David Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, 65:113–126, 1968.

[10] Paul F. Syverson. Towards a strand semantics for authentication logics. In *Electronic Notes in Theoretical Computer Science*, 20,2000.

[11] Paul F. Syverson and Paul C. van Oorschot. On unifying some cryptographic protocol logics. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 14–28. IEEE CS Press, May 1994.

[12] Paul F. Syverson and Paul C. van Oorschot. A unified cryptographic protocol logic. NRL Publication 5540-227, Naval Research Lab, 1996.

[13] M.-J. Toussaint and P. Wolper. Reasoning about cryptographic protocols. In J. Feigenbaum and M. Merritt, editors, *Distributed Computing and Cryptography*, volume 2 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 245–262. American Mathematical Society, 1989.

[14] Gabriele Wedel and Volker Kessler. Formal semantics for authentication logics. In E. Bertino, H. Kurth, and G. Martella, editors, *ESORICS'96*, LNCS 1146, pages 219–241. Springer-Verlag, 1996.

Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties *

Fabio Martinelli[1], Ilaria Matteucci[1,2]
Istituto di Informatica e Telematica - C.N.R., Pisa, Italy[1]
{Fabio.Martinelli, Ilaria.Matteucci}@iit.cnr.it
Dipartimento di Matematica, Università degli Studi di Siena[2]

Abstract

In this paper we show how the partial model checking approach for the analysis of secure systems may also be useful for enforcing security properties. We define a set of process algebra operators that act as programmable controllers of possibly insecure components. The program of these controllers may be automatically obtained through the usage of satisfiability procedures for a variant of μ-calculus.

1 Overview

Many approaches for the analysis of security properties have been successfully developed in the last two decades. An interesting one is based on the idea that potential attackers should be analyzed as if they were un-specified components of a system; thus reducing security analysis to the analysis of *open* systems [11, 12, 14].

More recently there has been also interest on mechanisms and techniques to enforce security properties. A notable example is the security automata in [19] and some extensions proposed in [9].

The paradigm of analysis of security as analysis of open systems has been extended to cope with security protocols [14], fault tolerance [7] and recently access control based on trust management [15]. In this paper we enrich this theory with a method for (automatically) enforcing several security properties.

Basically, we define a set of process algebra operators. They act as programmable controllers of a component that must be managed in order to guarantee that the overall system satisfies a given security policy. Also, we develop a technique to automatically synthesize the appropriate controllers. This represent a significant contribution w.r.t. to the previous work in [9, 19], where this issue was not addressed. The synthesis is based on a satisfiability procedure for the μ-calculus.

Moreover, under certain hypothesis on the observation power of the enforcing controllers, we are able to enforce some non-interference properties (for finite-state systems) that were not intentionally addressed in [19], due to the specific assumptions they had on the enforcing mechanisms.

Our logical approach is also able to cope with composition problems, that have been considered as an interesting and challenging issue in [3].

This paper is organized as follows. Section 2 recalls the basic theory about the analysis of security properties, especially non-interference as properties of open systems. Section 3 explains our approach and Section 4 extends it to manage several kinds of enforcement mechanisms. Section 5 illustrates an example. Section 6 presents a discussion on related work and eventually Section 7 concludes the paper.

*Work partially supported by CNR project "Trusted e-services for dynamic coalitions" and by a CREATE-NET grant for the project "Quality of Protection (QoP)". A full version of this paper with the proofs appears as Technical report of IIT-CNR [16].

2 Background

In this section we briefly recall some technical machinery used in our approach and also a logical approach for dealing with information flow properties (and security properties in general).

2.1 A language for describing concurrent and distributed systems

The *Security Process Algebra* (SPA) [6] is used to describe concurrent and distributed systems and is derived from CCS process algebra of R. Milner [17]. The syntax of SPA is the following:

$$E ::= \mathbf{0} \mid \alpha.E \mid E_1 + E_2 \mid E_1 \| E_2 \mid E \backslash L \mid Z$$

where α is an action in Act, $L \subseteq \mathcal{L}$ and Z is a process constant that must be associated with a definition $Z \doteq E$. As usual, constants are assumed to be *guarded* [17], i.e. they must be in the scope of some prefix operator $\alpha.E'$. The set of SPA *processes* (i.e., terms with guarded constants), is denoted with $\mathcal{E}$, ranged over by $E, F, P, Q \ldots$. We will often use some common syntactic simplifications, e.g., omission of trailing $\mathbf{0}$'s as well as omission of brackets on restriction on a single action. $Sort(E)$ is used to denote the set of actions that occurs in the term E.

SPA operators have the following informal meaning:

- $\mathbf{0}$ is a process that does nothing;
- $\alpha.E$ is a process that can perform an α action and then behaves as E;
- $E_1 + E_2$ (*choice*) represents the nondeterministic choice between the two processes E_1 and E_2;
- $E_1 \| E_2$ (*parallel*) is the parallel composition of two processes that can proceed in an asynchronous way , synchronizing on complementary actions, represented by an internal action τ, to perform a communication.
- $E \backslash L$ (*restriction*) is the process E when actions in $L \cup \overline{L}$ are prevented.

The operational semantics of SPA terms is given in terms of Labeled Transitions Systems (LTS).

Definition 2.1 *A labeled transition system $(\mathcal{E}, \mathcal{T})$ (LTS) of concurrent processes over Act has the process expressions $\mathcal{E}$ as its states, and its transitions $\mathcal{T}$ are exactly which can be inferred from the transition rules for processes.*

The interested reader may find the formal definition of the semantics below:

$$\frac{}{\alpha.E \xrightarrow{\alpha} E} \quad \frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \quad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 \| E_2 \xrightarrow{a} E_1' \| E_2} \quad \frac{E_2 \xrightarrow{a} E_2'}{E_1 \| E_2 \xrightarrow{a} E_1 \| E_2'} \quad \frac{E_1 \xrightarrow{l} E_1' \quad E_2 \xrightarrow{\bar{l}} E_2'}{E_1 \| E_2 \xrightarrow{\tau} E_1' \| E_2'}$$

$$\frac{Z \doteq E \quad E \xrightarrow{\alpha} E'}{Z \xrightarrow{\alpha} E'} \quad \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 \backslash L \xrightarrow{\alpha} E_1' \backslash L} (\alpha \notin L \cup \overline{L})$$

2.2 Strong and weak bisimulations

It is often necessary to compare processes that are expressed using different terms but have the same behavior. We recall some useful relations on processes.

Definition 2.2 *Let $(\mathcal{E}, \mathcal{T})$ be an LTS of concurrent processes, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* strong simulation *(denoted by $\prec$) over $(\mathcal{E}, \mathcal{T})$ if and only if, whenever $(E, F) \in \mathcal{R}$ we have:*

$$\text{if } E \xrightarrow{a} E' \text{ then there exists } F' \in \mathcal{E} \text{ s. t. } F \xrightarrow{a} F' \text{ and } (E', F') \in \mathcal{R}$$

Now, we can define *strong bisimulation*:

Definition 2.3 *A binary relation $\mathcal{R}$ over $\mathcal{E}$ is said a* strong bisimulation *(denoted by $\sim$) over the LTS of concurrent processes $(\mathcal{E}, \mathcal{T})$ if both $\mathcal{R}$ and its converse are strong simulation.*

Another kind of bisimulation is the *weak bisimulation*. This relation is used when there is the necessity of understanding if systems with different internal structure - and hence different internal behavior - have the same external behavior and may thus be considered observationally equivalent. The notion of *observational relations* is the follow: $E \stackrel{\tau}{\Rightarrow} E'$ (or $E \Rightarrow E'$) if $E \xrightarrow{\tau}{}^{*} E'$ (where $\xrightarrow{\tau}{}^{*}$ is the reflexive and transitive closure of the $\xrightarrow{\tau}$ relation); for $a \neq \tau$, $E \stackrel{a}{\Rightarrow} E'$ if $E \stackrel{\tau}{\Rightarrow} \xrightarrow{a} \stackrel{\tau}{\Rightarrow} E'$. Let $DerE$ be the set of derivatives of E, i.e., the set of process that can be reached through the transition relations. Now we are able to give the two following definitions.

Definition 2.4 *Let $\mathcal{R}$ be a binary relation over a set of process $\mathcal{E}$. Then $\mathcal{R}$ is said to be a* weak simulation *(denoted by $\precsim$) if, whenever $(E, F) \in \mathcal{R}$,*

$$\text{if } E \xrightarrow{a} E' \text{ then there exists } F' \in \mathcal{E} \text{ s. t. } F \stackrel{a}{\Rightarrow} F' \text{ and } (E', F') \in \mathcal{R}.$$

Definition 2.5 *A binary relation $\mathcal{R}$ over $\mathcal{E}$ is said a* weak bisimulation *($\approx$) over the LTS of concurrent processes $(\mathcal{E}, \mathcal{T})$ if both $\mathcal{R}$ and its converse are weak simulation.*

Every strong simulation is also a weak one (see [17]).

2.3 Equational μ-calculus

Modal μ-calculus is a process logic well suited for specification and verification of systems whose behavior is naturally described using state changes by means of actions. It is a normal modal logic K augmented with recursion operators. It permits to express a lot of interesting properties like *safety* and *liveness* properties, as well as allowing us to express equivalence conditions over LTS.

In equational μ-calculus recursion operators are replaced by fixpoint equations. This permits to recursively define the properties of a given systems.

We use the equational μ-calculus instead of modal μ-calculus because the former is very suitable for partial model checking, that is described later (see [1], [2]).

Let a be in Act and X be a variable ranging over a finite set of variables $Vars$.

Given the grammar:

$$A ::= X \mid \mathbf{T} \mid \mathbf{F} \mid X_1 \wedge X_2 \mid X_1 \vee X_2 \mid \langle a \rangle X \mid [a]X$$
$$D ::= X =_\nu AD \mid X =_\mu AD \mid \epsilon$$

where the meaning of $\langle a \rangle X$ is 'it is possible to do an a-action to a state where X holds' and the meaning of $[a]X$ is 'for all a-actions that are performed then X holds'. $X =_\nu A$ is a minimal fixpoint equation, where A is an assertion (i.e. a simple modal formula without recursion operator), and $X =_\mu A$ is a maximal fixpoint equation. Roughly, the semantic $[\![D]\!]$ of the list of equations D is the solution of the system of equations corresponding to

D. According to this notation, $[\![D]\!](X)$ is the value of the variable X, and $E \models D \downarrow X$ can be used as a short notation for $E \in [\![D]\!](X)$. The following result can be proved by putting together standard results for decision procedures for μ-calculus (see [20]).

Theorem 2.1 *Given a formula γ it is possible to decide in exponential time in the length of γ if there exists a model of γ and it is also possible to give an example of it.*

2.4 Partial model checking

Partial model checking (*pmc*) is a technique that was originally developed for compositional analysis of concurrent systems (processes) (see [2]). The intuitive idea underlying the pmc is the following: proving that $E\|F$ satisfies a formula ϕ is equivalent to prove that F satisfies a modified specification $\phi_{//_E}$, where $//_E$ is the partial evaluation function for the parallel composition operator(see [2]).In formula:

$$E\|F \models \phi \tag{1}$$

In order to describe how pmc function acts, we discuss, for instance, the partial evaluation rules for the formula $\langle\tau\rangle A$ w.r.t. the $\|$ operator. By inspecting the inference rules, we can note that the process $E\|F$ (with F unspecified component) can perform a τ action by exploiting one of the three possibilities:

- the process F performs an action τ going in a state F' and $E\|F'$ satisfies A; this is taken into account by the formula $\langle\tau\rangle(A_{//_E})$;
- the process E performs an action τ going in a state E' and $E'\|F$ satisfies A and this is considered by the disjunctions $\bigvee_{E \xrightarrow{\tau} E'} A_{//_{E'}}$, where every formula $A_{//_{E'}}$ takes into account the behavior of F in composition with a τ successor of E;
- the last possibility is that the τ action is due to the performing of two complementary actions by the two processes. So for every a-successor E' of E there is a formula $\langle\bar{a}\rangle(A_{//_{E'}})$.

With partial model checking we can reduce the previous property to:

$$F \models \phi_{//E} \tag{2}$$

Lemma 2.1 *Given a process $E\|F$ and a formula ϕ we have:*

$$E\|F \models \phi \textit{ iff } F \models \phi_{//E}$$

A similar lemma holds for every operator of SPA (see [1]).

In this way, it can be noticed that the reduced formula $\phi_{//E}$ depends only on the formula ϕ and on process E. No information is required on the process F which can represent a possible enemy. Thus, given a certain system E, it is possible to find the property that the enemy must satisfy in order to make a successful attack on the system. It is worth noticing that partial model checking functions may be automatically derived from the semantics rules used to define a language semantics (Structured Operational Semantics). Thus, the proposed technique is very

flexible. Here, we give the *pmc* function for parallel operator (that can be also found in [1, 2]).

$$\begin{aligned}
(D{\downarrow}X)_{//t} &= (D_{//t}){\downarrow}X_t \\
\epsilon_{//t} &= \epsilon \\
(X =_\sigma AD)_{//t} &= ((X_s =_\sigma A_{//s})_{s\in Der(t)})(D)_{//t} \\
X_{//t} &= X_t \\
[a]\,A_{//s} &= [a](A_{//s}){\wedge}\textstyle\bigwedge_{s\xrightarrow{a}s'} A_{//s'} \text{ if } a{\neq}\tau \\
[\tau]\,A_{//s} &= [\tau](A_{//s}){\wedge}\textstyle\bigwedge_{s\xrightarrow{\tau}s'} A_{//s'}{\wedge}\bigwedge_{s\xrightarrow{a}s'}[\bar{a}](A_{//s'}) \\
(A_1{\wedge}A_2)_{//s} &= ((A_1)_{//s}){\wedge}((A_2)_{//s}) \\
\mathbf{T}_{//s} &= \mathbf{T}
\end{aligned}$$

2.5 Characteristic formulae

A *characteristic formula* is a formula in equational μ-calculus that completely characterizes the behavior of a (state in a) state-transition graph modulo a chosen notion of behavioral relation. It is possible to define the notion of characteristic formula for a given finite state process E w.r.t. weak bisimulation as follows (see [18]).

Definition 2.6 *Given a finite state process E, its characteristic formula (w.r.t. weak bisimulation) $D_E \downarrow X_E$ is defined by the following equations for every $E' \in Der(E)$, $a \in Act$:*

$$X_{E'} =_\nu \Big(\bigwedge_{a;E'':E'\xrightarrow{a}E''} \langle\langle a\rangle\rangle X_{E''}\Big) \wedge \Big(\bigwedge_{a}([a]\Big(\bigvee_{E'':E'\stackrel{a}{\Rightarrow}E''} X_{E''}\Big)\Big)$$

where $\langle\langle a\rangle\rangle$ of the modality $\langle a\rangle$ which can be introduce as abbreviation (see [18]):

$$\langle\langle\epsilon\rangle\rangle\phi \stackrel{def}{=} \mu X.\phi \vee \langle\tau\rangle X \quad \langle\langle a\rangle\rangle\phi \stackrel{def}{=} \langle\langle\epsilon\rangle\rangle\langle a\rangle\langle\langle\epsilon\rangle\rangle\phi$$

The following lemma characterizes the power of these formulae.

Lemma 2.2 *Let E_1 and E_2 be two different finite-state processes. If ϕ_{E_2} is characteristic for E_2 then:*

1. *If $E_1 \approx E_2$ then $E_1 \models \phi_{E_2}$*
2. *If $E_1 \models \phi_{E_2}$ and E_1 is finite-state then $E_1 \approx E_2$.*

2.6 A logical approach for specifying and analyzing information flow properties

Information flow is a main topic in the theoretical study of computer security. We can find several formal definitions in the literature (see [10]). To describe this problem, we can consider two users, *High* and *Low* interacting with the same computer system. We ask if there is any flow of information from *High* to *Low*. The central property is the *Non Deducibility on composition* (*NDC*, see [6]): the low level users cannot infer the behavior of the high level user from the system because for the low level users the system is always the same. This idea can be represented as follow:

$$\forall \Pi \in \text{ High users } E \mid \Pi \equiv E \text{ w.r.t. Low users}$$

(where $\mid$ represents a suitable composition operator.) We study this property in term of *SPA* parallel composition operator and *bisimulation* equivalence.

We denote with $BNDC$ a security property called *Bisimulation Non Deducibility on Compositions* (see [6]).

Definition 2.7 *Let $\mathcal{E}_H = \{\Pi \mid Sort(\Pi) \subseteq H \cup \{\tau\}\}$ be the set of High users. $E \in BNDC$ if and only if $\forall \Pi \in \mathcal{E}_H$ we have $(E\|\Pi)\backslash H \approx E\backslash H$.*

By using the characteristic formula ϕ of the process $E\backslash H$, we may express information flow property in a logical way.

$$E \in BNDC \text{ iff } \forall \Pi \in S : (E\|\Pi)\backslash H \models \phi \tag{3}$$

Partial model checking function gives we have a method for reducing the verification of the previous property to a validity checking problem in μ-calculus (see [11]). As a matter of fact, the property 4 turns out to be equivalent to

$$E \in BNDC \text{ iff } \forall \Pi \in S : \Pi \models \phi' \tag{4}$$

where ϕ' is the formula obtained from ϕ after *pmc* w.r.t the process E (and the restriction operator). Thus, due the decidability of the validity problem for μ-calculus we have.

Proposition 2.1 *$BNDC$ is decidable for all finite state processes E.*

Our logical approach has been extended to cope with several security properties. Thus the approach we are going to introduce is applicable to a wide set of security properties.

3 Our approach for enforcing security properties

Let S be a system, and let X be one component that may be dynamically changed (e.g., a downloaded mobile agent). We say that the system $S\|X$ enjoys a security property expressed by a logical formula ϕ if and only if for every behavior of the component X, the behavior of the system S enjoys that security property:

$$\forall X (S\|X)\backslash H \models \phi \tag{5}$$

where $H = Sort(X)$.

By using the partial model checking approach proposed in [12], we can focus on the properties of the possibly un-trusted component X, i.e.:

$$\forall X \quad X \models \phi_{S,\backslash H} \tag{6}$$

Thus, we may study whether a potential enemy could exists and, in particular, which are necessary and sufficient conditions that an enemy should satisfy for the purpose to alter the correct behavior of the system.

In order to protect the system we may simply check each process X before executing it or, if we do not have this possibility, we may define a controller that in any case forces it to behave correctly.

We may distinguish several situations[1] depending on the control one may have on the process X:

1. if X performs an action we may detect and intercept it;
2. in addition to 1), it is possible to know which are the possible next steps of X;
3. X whole code is known and we are able to model check it[2].

In the scenarios 1) and 2) we may imagine to develop some controllers that force the intruder to behave correctly, i.e. as prescribed by the formula $\phi_{S,\backslash H}$.

[1] The last two pose several decidability issues.

[2] We do not consider here the possibility of manipulate the code.

3.1 Enforcing security properties with programmable controllers

We wish to provide a framework where we are able to enforce specific security properties defining a new operator, say $Y \triangleright^* X$, that can permit to control the behavior of the component X, given the behavior of a control program Y.

Example 3.1 *Let E and F be two processes, and let $a \in Act$ be an action. We define a new operator $\triangleright'$ (controller operator) by these two rules:*

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F'} \tag{7}$$

$$\frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F} \tag{8}$$

This operator forces the system to make always the right action also if we do not know what action the agent X is going to perform.

Eventually, we would like that the overall system $S \| (Y \triangleright^* X)$ always enjoys the desired security properties regardless of the behavior of the component X. Thus, we want to find a control program Y such that:

$$\forall X (S \| Y \triangleright^* X) \backslash H \models \phi \tag{9}$$

Equivalently, by *pmc*, we get:

$$\exists Y \; \forall X \; (Y \triangleright^* X) \models \phi' \tag{10}$$

where $\phi' = \phi_{//(S,\backslash H)}$.

Note that differently from other approaches the control target and the controller are expressed in a similar formalism.

While the equation 10 should be the property to manage, it might not be easy. However, we note that if the controller operator satisfies the following additional property

Assumption 3.1 *For every X and Y, we have:*

$$Y \triangleright^* X \sim Y$$

then the property 10 is equivalent to:

$$\exists Y \; Y \models \phi' \tag{11}$$

As a matter of fact, the previous assumption permits us to conclude that $Y \triangleright^* X$ and Y are strongly equivalent on so they satisfy the same formulas. The formulation 11 is easier to be managed.

Refer to example 3.1, we are able to prove that the operator $\triangleright'$ enjoys Assumption 3.1.

Proposition 3.1 *The operator $\triangleright'$ enjoys Assumption 3.1.*

Note that for some properties, e.g. BNDC, it is sufficient that $Y \triangleright^* X$ and Y are weakly bisimilar. According to definition of weak bisimulation, $Y \triangleright^* X \approx X$ (since every strong simulation is also a weak one [17]) and thus it could be applied to enforce information flow properties (although in the scenario 1) it would not be very useful, since it could often override the high user instructions).

While designing such a process Y could not be difficult in principle, we can take advantage of our logical approach and obtain an automated procedure as follows.

3.2 Automated synthesis of controllers

In this subsection, we discuss how it is possible to find a program controller Y that is a model of ϕ', the formula in 11.

As a matter of fact, our logical approach is very useful.

The formula ϕ' is a μ-calculus formula, so, referring to the theorem 2.1, it is possible to decide if there exists a model of such ϕ'. The procedure returns also a model that will be our program for our controllers.

Unfortunately, the satisfiability procedure has complexity that is, in the worst case, exponential in the size of the formula.

3.3 Composition of properties

Our logical approach is able to struggle successfully with composition problems. If we should force many different security policies, we have only to force the conjunction of this policies. In formulas: let $\phi_1, \cdots, \phi_n$ be n different security policies, S be our system and X be an external agent, we have:

$$\forall X(S \| X) \backslash H \models \phi_1 \quad \ldots \quad \forall X(S \| X) \backslash H \models \phi_n$$

The following step to solve is reduce this n proposition to one in the following way:

$$\forall X(S \| X) \backslash H \models \bigwedge_{i=1,\cdots,n} \phi_i \tag{12}$$

If we assume $\bigwedge_{i=1,\cdots n} \phi_i = \phi$, we have the same situation that we have described by the formula 10.

4 Other controllers

We can define other controller operators as follows.

The controller $\triangleright''$ have two rules:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F'} \tag{13}$$

$$\frac{E \xrightarrow{a} E' \; F \overset{a}{\not\rightarrow}}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F} \tag{14}$$

This controller is the most complete: if the program E and the target F agree on the next action both can do it in a lock step, if F does not have a correct behavior, the process E issues an action, so the system maintains a correct behavior. Being able to give priorities to rule applications, definitely the first rule should have higher priority than the second one.

The following result holds.

Proposition 4.1 *The preposition 3.1 holds also for two operator: $\triangleright'$ and $\triangleright''$.*

Another interesting operator is described by the following rule:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright''' F \xrightarrow{a} E' \triangleright''' F'} \tag{15}$$

However, it is useful to note that for this operator a weaker proposition holds.

Proposition 4.2 *Between $Y \triangleright''' X$ and Y holds the following relations:*

$$Y \triangleright''' X \prec Y$$

i.e. $Y \triangleright''' X$ and Y are strong similar but not bisimilar.

As a matter of fact, with this operator, we can ensure that the system is secure only w.r.t. security properties that are safety properties. Such properties are preserved under weak simulation (e.g. see [7]). Thus, we cannot enforce liveness properties through this controller.

4.1 Feasibility issues for our controllers

The introduction of a controller operator helps to guarantee a correct behavior of the entire system.

We discuss in this subsection, how and also if, these controllers ($\triangleright'$, $\triangleright''$ and $\triangleright'''$) can be effectively implemented.

However, the actual feasibility of these controllers depends on the scenarios we consider. In particular, we focus on scenarios 1) and 2).

For the first controller operator, $\triangleright'$, we can note that this operator need to check the next action or it can directly execute one correct action. Thus, it would be easily implementable in all the two scenarios.

The operator $\triangleright''$ cannot be implemented in the scenario 1): if we cannot decide a priori which are the possible next steps that the external agent is not able to perform, we cannot implement the second rule (14). In the scenario 2), such an operator would be implementable. It would be also possible in the scenario 2), if we could also know whether X is forced to make a specific action, to give priority to the first rule in order to allow always the correct actions of the target. Thus, controller $\triangleright''$ would be the most appropriate in this scenario.

The last controller operator can be implemented in any scenarios. As a matter of fact, it coincides with the monitors defined in [19].

5 A simple example

Consider the process $E = l.\mathbf{0} + h.h.l.\mathbf{0}$. The system E where no high level activity is present is weakly bisimilar to $l.\mathbf{0}$.

Consider the following equational definition (please note that F is a variable here):

$$F =_\nu ([\tau]F) \wedge [l]\mathbf{T} \wedge \langle\langle l\rangle\rangle\mathbf{T}$$

It asserts that a process may and must perform the visible action l.

As for the study of $BNDC$-like properties we can apply the partial evaluation for the parallel operator we obtain after some simplifications:

$$F_E \quad =_\nu \quad ([\tau]F_E) \wedge [\overline{h}]\langle\langle\overline{h}\rangle\rangle\mathbf{T}$$

which, roughly, expresses that after performing a visible $\overline{h}$ action, the system reaches a configuration s.t. it must perform another visible $\overline{h}$ action.

The information obtained through partial model checking can be used to enforce a security policy which prevents a system from having certain information leaks. In particular, if we use the definition of the controller as $\triangleright''$, we simply need to find a process that is a model for the previous formula, say $Y = \overline{h}.\overline{h}.\mathbf{0}$.

Then, for any component X, we have $(E\|(Y \triangleright'' X)) \setminus \{h\}$ satisfies F.

For instance, consider $X = \overline{h}.\mathbf{0}$. The system

$$(E\|(Y \triangleright'' X)) \setminus \{h\} \xrightarrow{\tau} (h.l.\mathbf{0}\|(\overline{h} \triangleright'' \mathbf{0})) \setminus \{h\}$$

Thus, using the second rule the controller may force to issue another $\overline{h}$ and thus we eventually get

$$(h.l.\mathbf{0} \| (\overline{h} \rhd'' \mathbf{0})) \setminus \{h\} \xrightarrow{\tau} (l.\mathbf{0} \| (\mathbf{0} \rhd'' \mathbf{0})) \setminus \{h\} \approx l.\mathbf{0}$$

and so the system still preserve its security since the actions performed by the component X have been prevented from being visible outside. On the contrary, if the controller would not be present, there would be a deadlock after the first internal action.

6 Discussion on related work

In [13], we presented preliminary work based on different techniques for automatically synthesizing systems enjoying a very strong security property, i.e. SBSNNI (e.g., see [6]). That work did not deal with controllers.

Much of prior work is about the study of enforceable properties and related mechanisms.

In [19], Schneider deals with enforceable security properties in a systematic way. He discusses whether a given property is enforceable and at what cost. To study those questions, Schneider uses the class of enforceable mechanisms (EM) that work by monitoring execution steps of some system, herein called the *target*, and terminating the target's execution if it is about to violate the security property being enforced. The author asserts there isn't any EM (Execution Monitoring) that can enforce information flow because it can't be formalized like a safety property. The security automata defined in [19] have the follow behavior:

- If the automaton can make a transition on given input symbol, then the target is allowed to perform that step. The state of the automaton changes according to the transition rules.
- otherwise the target is terminated and we can deduce that security property can be violated.

He explicitly assumes to be in the scenario that we call 1).

We can note that our controller operator, $\rhd'''$, have the same behavior of the security automata for enforcement that Schneider defines in his article.

The operator $\rhd'''$ have only the following rule:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd''' F \xrightarrow{a} E' \rhd''' F}$$

Roughly speaking, if process F does the correct action then $E \rhd''' F$ does a correct transaction else the system stops.

This fact is very important because, as we say in the proposition 4.2, $Y \rhd''' X$ and Y are strongly similar but not bisimilar. So this two processes are not strongly equivalent and they don't satisfy all the same formulas. So, also with our formalism, we can not enforce information flow with this operator.

We can however define an operator in scenario 1) that enforces information flow property. The cost of this operation is that the behavior of the controller component may be completely neglected. Thus, from a practical point of view, our operator is not very useful.

However, we may notice that our work is a contribution w.r.t. the work of Schneider since it allows the automatic construction of the correct monitor.

Also in [3, 9] there is the idea that information flow can not be forced by an automaton. In both of these articles, many types of automata are illustrated. All of them are in the scenario 1). The automata waits for an action of the target. In particular, in [9] there are four different automata:

truncation automata it can recognize bad sequences of actions and halt program execution before the security property is violated, but cannot otherwise modify program behavior. These automata are similar to Schneider's original security monitor;

suppression automata in addition to being able to halt program execution, it has the ability to suppress individual program actions without terminating the program outright;

insertion automata it is able to insert a sequence of actions into the program action stream as well as terminate the program;

edit automata it combines the powers of suppression and insertion automata. It is able to truncate action sequences and insert or suppress security-relevant actions at will.

The interested reader may find in the full version of this paper (see [16]) the description of process algebras operators that mimic as such automata. (Since that truncation automata is the same automata is described in [19], we already defined a controller operator which has the same behavior.)

We use controller synthesis in order to force a system to verify security policy. The synthesis of controllers is, however, studied also in other research areas. We describe here two papers that deal with synthesize of controller in real-time.

In [4] the author describes an algorithm for synthesize controller from real-time specification. He presents an algorithm for specified in a subset of the internal temporal logic Duration calculus. The synthesized controllers are given as PLC-Automata. These are an abstract representation of a machine that periodically polls the input and has the possibility of measuring time.

In [5] the authors tackles the following problem: given a timed automaton restrict its transition relation in a systematic way so that all remaining behaviors satisfy certain properties. The problem is formulated using the notion of real-time game. A strategy for a given game is a rule that tells the controller how to choose between several possible actions in any game position.A strategy is winning if the controller, by following these rules, always wins (according to a given definition of winning) no matter what the environment does. There is the definition of Game automata and the authors gives a relation and using this relation is able to define a winning strategy for the game.

7 Conclusion and future work

We illustrated some preliminary results towards a uniform theory for enforcing security properties. With this work, we contribute to extend a framework based on process calculi and logical techniques that have been shown to be very suitable to model and verify several security properties. With respect to prior work, we also add the possibility to automatically build enforcing mechanisms.

Much work need to be done in order to make our approach more feasible in practice. We argue that there are many security properties whose corresponding controller may be built more efficiently. For instance, there are some cases in which the complexity of satisfiability problem is linear in the size of the formula (e.g., see [8]).

We argue that extending our approach to consider timed security properties should be possible and worth of investigation.

8 Acknowledgement

We thank the anonymous referees of FCS05 for valuable comments that helped us to improve this paper.

References

[1] H. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, June 1993.

[2] H. R. Andersen. Partial model checking. In *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 398. IEEE Computer Society, 1995.

[3] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In I. Cervesato, editor, *Foundations of Computer Security: proceedings of the FLoC'02 workshop on Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002. DIKU Technical Report.

[4] H. Dierks. Synthesising controllers from real-time specifications. In *ISSS '97: Proceedings of the 10th international symposium on System synthesis*, pages 126–133, Washington, DC, USA, 1997. IEEE Computer Society.

[5] A. P. E. Asarin, O. Maler and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. Elsevier, 1998.

[6] R. Focardi and R.Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1):5–33, 1997.

[7] S. Gnesi, G. Lenzini, and F. Martinelli. Logical specification and analysis of fault tolerant systems through partial model checking. *International Workshop on Software Verification and Validation (SVV), ENTCS.*, 2004.

[8] D. Janin and I. Walukiewicz. Automata for the modal μ-calculus and related results. In *Proc. of the 20th International Foundations of Computer Science 1995 (MFCS)*, pages 552–5662, Prague, 1995.

[9] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[10] G. Lowe. Semantic models for information flow. *Theor. Comput. Sci.*, 315(1):209–256, 2004.

[11] F. Martinelli. *Formal Methods for the Analysis of Open Systems with Applications to Security Properties*. PhD thesis, University of Siena, Dec. 1998.

[12] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*, page 44. IEEE Computer Society, 1998.

[13] F. Martinelli. Towards automatic synthesis of systems without informations leaks. In *Proceedings of Workshop in Issues in Theory of Security (WITS)*, 2000.

[14] F. Martinelli. Analysis of security protocols as *open* systems. *Theoretical Computer Science*, 290(1):1057–1106, 2003.

[15] F. Martinelli. Towards an integrated formal analysis for security and trust. *FMOODS 2005*, LNCS 3535, 2005.

[16] F. Martinelli and I. Matteucci. Partial model checking, process algebra operators and satisfiability procedures for (automatically) enforcing security properties. Technical report, IIT-CNR, March 2005.

[17] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[18] M. Müller-Olm. Derivation of characteristic formulae. In *MFCS'98 Workshop on Concurrency*, volume 18 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science B.V., August 1998. 12 pages, MFCS'98 Workshop on Concurrency.

[19] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[20] R. S. Street and E. A. Emerson. An automata theoretic procedure for the propositional μ-calculus. *Information and Computation*, 81(3):249–264, 1989.