

FCS 2013
Workshop on Foundations
of Computer Security
(Informal Proceedings)

June 29, 2013
Tulane University
New Orleans, Louisiana, USA
Affiliated with LICS'13 and CSF'13

Editors: Bruno Blanchet and Michael Clarkson

Preface

Welcome to the 10th Workshop on Foundations of Computer Security (FCS). This edition of the Workshop is held at Tulane University in New Orleans, Louisiana, USA, in affiliation with the 28th ACM–IEEE Symposium on Logic in Computer Science (LICS) and with the 26th IEEE Computer Security Foundations Symposium (CSF). We thank the organizers of those symposia for their assistance in organizing FCS, especially Patricia Bouyer-Decitre and Stephen Chong. We also thank Boris Köpf for his contribution of an invited talk at this year’s Workshop. And we gratefully acknowledge the members of the program committee for their work in assembling the program.

FCS originates from the Workshops on Formal Methods and Security Protocols (FMSP) in 1998 and 1999, and the Workshop on Formal Methods and Computer Security (FMCS) in 2000. The first FCS was held in 2002. The workshop continued to be held annually 2002–2010, sometimes in conjunction with other workshops (ARSPA, WITS, and PrivMod). FCS returns in 2013 after a two-year absence.

FCS 2013 is held in conjunction with the Workshop on Formal and Computational Cryptography (FCC). The two workshops were organized and reviewed separately but are meeting together on the same day. None of the FCC papers are included in this collection.

Out of 15 submitted papers, the FCS program committee accepted 9 for presentation at the workshop, with an acceptance rate of 60%. All papers received at least 3 reviews. This year FCS permitted one-page extended abstracts as submissions. These received the same review process as full-length submissions. Of the 15 submitted papers, 5 were extended abstracts; 4 of those were accepted. The abstract of Boris Köpf’s invited talk is also included in this collection.

FCS 2013 has no published proceedings. This unofficial collection of papers is intended for distribution to workshop participants; it should not be construed as an official proceedings. In particular, inclusion of a paper in this collection should not preclude submission to or publication in other venues.

June 2013

Bruno Blanchet
Michael Clarkson
FCS 2013 Co-Chairs

Program Committee

Myrto Arapinis (*University of Birmingham, UK*)
Aslan Askarov (*Harvard University, USA*)
Bruno Blanchet, co-chair (*INRIA Paris-Rocquencourt, France*)
Michael Clarkson, co-chair (*George Washington University, USA*)
Sara Foresti (*Università degli Studi di Milano, Italy*)
Deepak Garg (*Max Planck Institute for Software Systems, Germany*)
Catalin Hritcu (*University of Pennsylvania, USA*)
Alan Jeffrey (*Alcatel-Lucent Bell Labs, USA*)
Peeter Laud (*Cybernetica AS, Estonia*)
Gurvan Le Guernic (*DGA Maîtrise de l'Information, France*)
Stephen Magill (*IDA Center for Computing Sciences, USA*)
David Naumann (*Stevens Institute of Technology, USA*)
Andrei Sabelfeld (*Chalmers University of Technology, Sweden*)
Santiago Zanella Bèguelin (*Microsoft Research Cambridge, UK*)

Steering Committee

Martín Abadi (*Microsoft Research Silicon Valley and University of California, Santa Cruz, USA*)
Véronique Cortier (*LORIA, CNRS, France*)
John C. Mitchell (*Stanford University, USA*)
Andrei Sabelfeld (*Chalmers University of Technology, Sweden*)
Andre Scedrov (*University of Pennsylvania, USA*)
Vitaly Shmatikov (*University of Texas at Austin, USA*)
Luca Viganò, chair (*Università di Verona, Italy*)

Table of Contents

Invited Talk

Static Analysis of Cache Side Channels	1
<i>Boris Köpf</i>	

Submitted Papers

When Not All Bits Are Equal: Incorporating “Worth” into Information-flow Measures	2
<i>Mario S. Alvim, Andre Scedrov and Fred B. Schneider</i>	
Abstract Channels, Gain Functions and the Information Order	17
<i>Annabelle McIver, Carroll Morgan, Larissa Meinicke, Geoffrey Smith and Barbara Espinoza</i>	
MAP-REDUCE Enforcement Framework of Information Flow Policies . . .	18
<i>Minh Ngo, Fabio Massacci and Olga Gadyatskaya</i>	
A Framework for Composing Security-typed Languages	34
<i>Andreas Gampe and Jeffery Von Ronne</i>	
A Formal Framework for Secure Routing Protocols	49
<i>Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou and Boon Loo</i>	
Translating between Equational Theories for Automated Reasoning	50
<i>Ben Smyth, Myrto Arapinis and Mark Ryan</i>	
Using Interpolation for the Verification of Security Protocols (Extended Abstract)	51
<i>Giacomo Dalle Vedove, Marco Rocchetto, Luca Viganò and Marco Volpe</i>	
Bounded Memory Protocols and Progressing Collaborative Systems	52
<i>Max Kanovich, Tajana Ban Kirigin, Vivek Nigam and Andre Scedrov</i>	
A Multi-Role Translation of Protocol Narration into the Spi-Calculus with Correspondence Assertions	68
<i>Eijiro Sumii and Yuji Sato</i>	

Static Analysis of Cache Side Channels

Boris Köpf

IMDEA Software Institute

Abstract. Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time or power consumption. CPU caches are a particularly rich source of side channels because their behavior heavily impacts the execution time of programs and can be monitored in various ways.

With the recent progress in automation of quantitative information-flow analysis, the formal analysis of cache side channels becomes feasible. We demonstrate this by building CacheAudit, a versatile platform for the automatic, static analysis of cache side channels based on abstract interpretation. CacheAudit takes as input a program binary and a cache configuration, and it derives formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, namely those based on observing cache states, traces of hits and misses, and execution times.

In this talk, I will present the foundations and architecture of the CacheAudit platform, and the results we obtain when analyzing library implementations of symmetric cryptosystems such as AES or Salsa. I will conclude with an outlook on how CacheAudit can be used for engineering certified proofs of security of leakage-resilient cryptosystems on platforms with concurrency and caches.

When not all bits are equal: Incorporating “worth” into information-flow measures

Mário S. Alvim¹, Andre Scedrov¹, and Fred B. Schneider²

¹ University of Pennsylvania, USA

² Cornell University, USA

Abstract. Approaches to quantitative information flow (QIF) traditionally have presumed that all leaks involving a given number of bits are equally harmful. The presumption is unrealistic, so a new approach to QIF is described. Here, secrets are defined in terms of fields, where derived secrets obtained by combining these fields can be assigned a different “worth” (perhaps in proportion to the harm that would result from disclosure). New measures that incorporate worth into QIF are then defined; they generalize probability of guessing, guessing entropy, and Shannon entropy. A lattice of information is derived to provide an underlying algebraic structure for an adversary’s state of knowledge in this more-general setting. **Keywords:** information-flow, quantitative methods, information theory, lattice of information, entropy.

1 Introduction

Quantitative information flow (QIF) is concerned with measuring how much information about a system’s secrets is being leaked to an adversary. The adversary is presumed to have *a priori information* about the secrets before execution starts and to access *public observables* as execution proceeds. By combining a priori information and public observables, the adversary achieves *a posteriori information* about the secrets. The *leakage* from an execution is then computed either (i) as the difference between a posteriori information and a priori information or, equivalently, (ii) as the difference between a priori uncertainty and a posteriori uncertainty (since knowledge of information is the dual of uncertainty).

This definition of leakage depends on how information (or uncertainty) is measured. Cachin [1] advocates that such definitions not only include a way to calculate some numeric value but also offer an *operational interpretation*, which describes what question the information measure answers in specified scenarios of interest. Popular definitions of information include:

- *probability of guessing* [2, 3], which measures how likely it is that the secret will be correctly inferred in a certain number of guesses,
- *guessing entropy* [4, 5], which measures how many tries are required before the secret will be correctly guessed, and

- *Shannon entropy* [6–12], which measures how much information is leaked per guess.

All of these definitions, however, treat secrets as monolithic.

In practice, secrets have structure. We can model this structure by viewing secrets as partitioned into *fields*, which are combined to form *derived secrets*. Since leakage of different derived secrets might cause different harms, a *worth assignment* is introduced to associate a *worth* with each derived secret. For instance, the secret corresponding to a client’s bank account might comprise two 7-digit derived secrets: a pin-code and a telephone number. Leaking the pin-code has the potential to cause considerable harm, so that derived secret would be assigned high worth; the telephone number would generally be public information, so this derived secret is assigned low worth.

Prior work in QIF has paid little attention to structure of secrets, implicitly assuming that all derived secrets are equally sensitive. This assumption can lead to misleading comparisons between systems that leak derived secrets with different worths but the same numbers of bits. The bank account example above can illustrate. Compare a system that leaks the pin-code with another that leaks the phone number—the same number of bits are leaked in both cases, but with rather different consequences.

Ignoring the structure of secrets also leads to misleading conclusions about harm from leaking different numbers of bits. Consider two systems that differ in the way they represent a house address. In system C_1 , standard postal addresses are used (i.e., a number, street name, and zip code); system C_2 uses GPS coordinates (i.e., a latitude and a longitude, each a signed 7-digit numbers). Under Shannon entropy with plausible sizes³ for address fields, C_1 requires 129 bits to represent a location that C_2 represents using 49 bits. Yet the same content is revealed whether C_1 leaks its 129 bits or C_2 leaks its 49 bits. (Of course the a priori information for addresses in C_1 is not zero, since certain values for a house number, street name, and zip code can be ruled out. And a similar argument can be made for C_2 , given knowledge of habitable terrain. Accounting for idiosyncrasies in the syntactic representation of secrets, however, can be a complicated task, hence an opportunity to introduce analysis mistakes. Worth assignments avoid some of that complexity.⁴)

This paper proposes an approach to QIF in which measures of information incorporate the structure of secrets and the worth of derived secrets. We call these measures *W-measures*. As in other QIF literature, we assume the adversary per-

³ Specifically, we assume a 4-digit house number, a 20-character alphabetic street name, and a 5-digit zip code.

⁴ Shannon [13] noted the problems that different representations can bring when he argues that the measure of information content should not depend on the syntax (“translation” in his terminology) used to represent this information: *These translations may be viewed as different ways of describing the same information in about the same way that a vector may be described by its components in various coordinate systems. The information itself may be regarded as the equivalence class of all translations or ways of describing the same information.*

forms attacks by controlling the low input to a deterministic system execution and observing the low outputs. And an attack is modeled as inducing partitions on the space of secrets, based on what the attacker observes. This characterization admits W -measures for the information contained in each partition; leakage is then defined as the difference in information between two partitions. By taking into account the worth assignment for derived secrets and the probabilistic distribution on secrets, we give measures for meaningful operational scenarios. In particular, our approach generalizes probability of guessing, guessing entropy, and Shannon entropy to the more expressive model that admits non-trivial worth assignments. Yet our methods remain consistent with the Lattice of Information (LoI) [14] that has provided an underlying algebraic structure for models involving sets of system executions.

The rest of the paper is structured as follows. Section 2 incorporates the structure and worth of secrets into a model for deterministic systems and attacks given in [15]. W -measures are covered in Section 3. In Section 4, the Lattice of Information is presented as an algebraic model for sets of possible attack sequences an adversary can perform; W -measures and leakage of information have natural connections to this lattice. Section 5 discusses related work, and our contributions are summarized in Section 6.

Remark: The corresponding technical report [16] contains full proofs, and examples comparing our W -measures with the measures they generalize.

2 A worth-based QIF model for deterministic systems

2.1 Deterministic systems and attacks

We follow a model for deterministic systems and attacks given by Köpf and Basin [15]. A *deterministic computational system*, or simply a *system*, is a function $\mathcal{C} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{O}$, where \mathcal{S} is a set of secrets, \mathcal{A} is a set of attacks, and \mathcal{O} is the set of observables that the system can produce. In an execution of \mathcal{C} , the high input is a secret $s \in \mathcal{S}$ chosen at the beginning of the computation according to a probability distribution $p_{\mathcal{S}}$, and the value of s is kept hidden from the adversary. The adversary can, however, choose an attack $a \in \mathcal{A}$ as a low input to \mathcal{C} . The system then computes $\mathcal{C}(s, a) = o$, producing the observable behavior $o \in \mathcal{O}$. Figure 1 depicts this. The nature of sets \mathcal{S} , \mathcal{A} and \mathcal{O} depends on the scenario of interest.

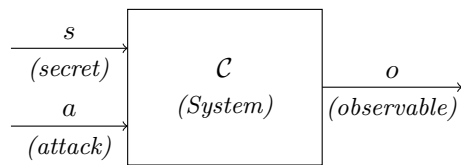


Fig. 1. A system with one high input, one low input, and one low output.

Example 1. A cryptographic system encodes messages using a secret key. The adversary can submit a message (e.g., a plain text) to the system and measure how long the encryption takes. So set \mathcal{S} of secrets corresponds to the set of all possible keys, set \mathcal{A} of attacks corresponds to messages the adversary can

choose from, and the set \mathcal{O} of observables is the range of time measurements the adversary can perform.

Since system \mathcal{C} is deterministic, each attack $a \in \mathcal{A}$ induces a partition⁵ P_a on the set of secrets. One block in the partition corresponds to each observable. Block $\mathcal{S}_{a,o} \in P_a$ contains all the secrets that are mapped to o when the low input to the system is a , i.e., $\mathcal{S}_{a,o} = \{s \in \mathcal{S} | \mathcal{C}(s, a) = o\}$. We omit the subscript corresponding to the attack when this is clear from the context, writing only \mathcal{S}_o for $\mathcal{S}_{a,o}$. We can identify an attack with the partition it induces. So once the attack sequence is fixed, the actual observation made by the adversary consists of learning the block to which the secret belongs. An attack step, thus, can be described mathematically as $\mathcal{C}(s, a) \in P_a$, which is a two-phase process:

1. the adversary chooses a partition P_a on \mathcal{S} , corresponding to attack $a \in \mathcal{A}$;
2. the system responds with the block $\mathcal{S}_o \in P_a$ that contains the secret.

Example 2. The adversary submits a message $a \in \mathcal{A}$ to the system of Example 1. He observes the elapsed time t required for encryption using the secret key, grouping into block \mathcal{S}_t all the keys that take time t to encrypt this particular message. Partition P_a on \mathcal{S} induced by the attack a is the collection of all blocks \mathcal{S}_t where $t \in \mathcal{O}$ is a time measurement.

The adversary may perform multiple attack steps while the same secret is maintained. An *attack sequence* is a list $\hat{a} = a_{t_1}, \dots, a_{t_k}$ of k attacks, where each $a_{t_i} \in \mathcal{A}$ represents the attack chosen at time t_i . Execution of an attack sequence starts when the secret s is chosen at the beginning of the computation, and execution ends either when the adversary does not perform further attacks or when the secret changes.

Example 3. Consider secret space $\mathcal{S} = \{1, 2, 3, 4, 5\}$, and two attacks $P_{a_1} = \{\{1, 2, 3\}, \{4, 5\}\}$ and $P_{a_2} = \{\{1, 3, 5\}, \{2, 4\}\}$. Suppose the adversary performs P_{a_1} and obtains an observable indicating that the secret belongs to the block $\{1, 2, 3\}$. In the sequence, the adversary performs attack P_{a_2} and learns that the secret belongs to the block $\{1, 3, 5\}$. The adversary can combine knowledge obtained by the two attack steps and deduce that the secret must belong to the set $\{1, 3\} = \{1, 2, 3\} \cap \{1, 3, 5\}$.

The adversary combines information acquired in an attack sequence by intersecting the partitions corresponding to each attack step, thereby obtaining a finer partition. So an attack sequence \hat{a} induces a partition $P_{\hat{a}} = \bigcap_{a \in \hat{a}} P_a$. Note, the partition induced by an attack sequence is independent of the order of its attack steps, so when indexing partitions we represent attack sequences as sets, rather than ordered lists.

An attack sequence of k steps produces as observables a tuple belonging to \mathcal{O}^k , i.e., a sequence $\mathcal{C}(s, a_{t_1}), \dots, \mathcal{C}(s, a_{t_k})$. However, since observables are

⁵ A *partition* P on a set \mathcal{S} divides \mathcal{S} into non-overlapping subsets. Formally, $P = \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ is a partition on \mathcal{S} iff: (i) $\bigcup_{\mathcal{S}_i \in P} \mathcal{S}_i = \mathcal{S}$; and (ii) for $1 \leq i \neq j \leq n$, $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$. Each $\mathcal{S}_i \in P$ is called a *block* in the partition.

relevant only when they identify blocks within a partition, an attack sequence \hat{a} can be seen as a single attack where the adversary chooses the partition $P_{\hat{a}}$ as the low input to the system, and then obtains as an observable the block the secret s belongs to. Formally, $\mathcal{C}(s, \hat{a}) \in P_{\hat{a}}$.

2.2 Measuring the information of attacks

As it is usual in QIF, we assume that the adversary knows probability distribution p_S on the set of secrets and function C describing system behavior. We also assume the adversary controls low inputs to the system—that is, the attack. The adversary’s goal is to infer as much information as possible from the secret, given knowledge about how the system works, attacks fed to the system, and observations made as the system executes.

High input to the system (i.e., the secret) is modeled as a random variable S . Following [5], partition $P_{\hat{a}} = \{\mathcal{S}_{o_1}, \dots, \mathcal{S}_{o_n}\}$ on \mathcal{S} induced by the attack sequence \hat{a} can be seen as a random variable with carrier $\{\mathcal{S}_{o_1}, \dots, \mathcal{S}_{o_n}\}$ and probability distribution $p_S(\mathcal{S}_{o_i}) = \sum_{s \in \mathcal{S}_{o_i}} p_S(s)$. We thus quantify information corresponding to an attack sequence, as follows. Let H_* be a generic measure of information (or uncertainty) of probability distributions. A priori information (or uncertainty) the adversary has about the secret is given by $H_*(p_S)$, and a posteriori information (or uncertainty) given a partition $P_{\hat{a}}$ is $H_*(p_S|P_{\hat{a}})$. So leakage of information is their difference.

The way of measuring the information (or uncertainty) contained in a partition depends on the operational interpretation under consideration. The traditional measures of probability of guessing, guessing entropy, and Shannon entropy ignore the structure and worth of secrets; Section 3 extends them to eliminate this limitation.

2.3 Modeling the structure and worth of secrets

We capture the structure of secrets by decomposing them into a collection of *fields*, each a piece of information with a domain. We use $\mathcal{F} = \{f_1, \dots, f_m\}$ to denote the (finite) set of fields in some scenario of interest. For $1 \leq i \leq m$, let $\text{domain}(f_i)$ be the domain of symbols for field f_i . A *secret* s is a tuple $s = \langle s[f_1], \dots, s[f_m] \rangle$, where $s[f_i] \in \text{domain}(f_i)$ is the symbol being stored in field f_i in secret s . So the set of possible secrets is $\mathcal{S} = \text{domain}(f_1) \times \dots \times \text{domain}(f_m)$.

Example 4. A bank system stores data about customer accounts. The secret account comprises fields $\mathcal{F} = \{\text{account number}, \text{pincode}, \text{customer}, \text{street number}, \text{street name}, \text{zipcode}\}$. We might have $\text{domain}(\text{pincode}) = \{1, \dots, 9999\}$ for the set of possible 4-digit pincodes, $\text{domain}(\text{street number}) = \{1, \dots, 9999\}$ corresponding to possible street numbers, and $\text{domain}(\text{street name}) = \{A, \dots, Z\}^*$ for the set of street names. Thus, a possible secret is $s = \langle \text{“ACC001”, “4576”, “JANE DOE”, “3300”, “WALNUT ST”, “19104”} \rangle$. In this case, $s[\text{pincode}] = \text{“4576”}$, $s[\text{street number}] = \text{“3300”}$, and $s[\text{street name}] = \text{“WALNUT ST”}$.

Subsets of the fields that constitute a secret often carry some valuable piece of information in their own. A *derived secret* of a secret is a tuple constructed from a subset \mathfrak{f} of fields \mathcal{F} . The set of all possible derived secrets is power set $\mathcal{P}(\mathcal{S})$ of fields. And for any particular subset $\mathfrak{f} = \{f_{i_1}, \dots, f_{i_k}\}$ of fields, the set of corresponding derived secrets is $\mathcal{S}[\mathfrak{f}] = \text{domain}(f_{i_1}) \times \dots \times \text{domain}(f_{i_k})$.

Example 5. Fields `street number`, `street name` and `zipcode` used by the bank system of Example 4 together form a derived secret `customer address` having domain $\text{domain}(\text{customer address}) = \text{domain}(\text{street number}) \times \text{domain}(\text{street name}) \times \text{domain}(\text{zipcode})$. So we have $s[\text{customer address}] = \langle \text{"3300"}, \text{"WALNUT ST"}, \text{"19104"} \rangle$.

A *worth assignment* attributes to each derived secret a non-negative, real number. Worth can be seen as the utility obtained by an attacker who learns the contents of the derived secret. Presumably, this value is proportional to the damage suffered should the contents of the derived secret become known to the adversary.

Definition 1. A worth assignment is a function $\omega : \mathcal{P}(\mathcal{F}) \rightarrow \mathbb{R}$ from the set of derived secrets to reals, satisfying for all $\mathfrak{f}, \mathfrak{f}' \in \mathcal{P}(\mathcal{F})$:

- (i) non-negativity: $\omega(\mathfrak{f}) \geq 0$;
- (ii) monotonicity: $\mathfrak{f} \subseteq \mathfrak{f}' \implies \omega(\mathfrak{f}) \leq \omega(\mathfrak{f}')$.

We require non-negativity of ω because a derived secret cannot carry a negative amount of information; we require monotonicity because every set of fields should be at least as sensitive as any of its subsets. Monotonicity implies that the worth of the set of fields as a whole, $\omega(\mathcal{F})$, is an upper bound for the worth of every derived secret. Here we will define the worth of a derived secret independently of its contents (e.g., the derived secret corresponding to a bank account has the same worth independently of whose bank account it is).

Example 6. Table 1 shows a possible worth assignment for some derived secrets of the secret `account` from Example 4. Notice that $\omega(\text{pincode}) = 0.50$ and $\omega(\text{zipcode}) = 0.10$, implying that the pin-code is more sensitive than zipcode. Also, observe that `account number` and `pincode` are assigned, together, more worth than their sum.

Derived secret \mathfrak{f}	Worth $\omega(\mathfrak{f})$
The whole secret	1.00
{account number}	0.10
{pincode}	0.50
{account number, pincode}	0.90
{street name}	0.01
{zipcode}	0.10
{street number}	0.05
{street number, street name, zipcode}	0.30

Table 1. Worth assignment for some derived secrets in Example 6.

2.4 W -measures

The adversary's knowledge about the secret depends on the partition induced by an attack sequence. Therefore, to quantify information (and leakage), we need to define measures of the information contained in a partition.

The first step is to define the worth of a secret as a proxy for how useful knowledge of that secret would be to the adversary. Define *worth of a secret* $s \in \mathcal{S}$ to be the worth of learning all of its fields, i.e., $\omega(s) = \omega(\mathcal{F})$.

Definition 2. Given a set \mathcal{S} , let Ω be the set of all possible worth assignments for the derived secrets of \mathcal{S} . Let $P_{\mathcal{S}}$ be the set of all probability distributions on \mathcal{S} , and $LoI(\mathcal{S})$ be the set of all partitions on \mathcal{S} . A W -measure is a function $\nu : \Omega \times P_{\mathcal{S}} \times LoI(\mathcal{S}) \rightarrow \mathbb{R}^+$.

The quantity $\nu(\omega, p_{\mathcal{S}}, P_{\hat{a}})$ represents the information (or uncertainty) w.r.t. \mathcal{S} contained in partition $P_{\hat{a}}$, given probability distribution $p_{\mathcal{S}}$ on secrets and worth assignment ω . What the W -measure quantifies depends on its operational interpretation. A W -measure that quantifies the worth of a partition, i.e., the utility it provides to the adversary, is called a *WW-measure*. Other W -measures may instead quantify some aspect linked to the extraction of some worth: a *WP-measure* quantifies the probability of the adversary achieving that worth; whereas a *WN-measure* quantifies the expected number of guesses needed to do so.

We pay particular attention to a class of W -measures on partitions whose value can be calculated as a function of the information of each of its blocks. Intuitively, given a W -measure ν , the information contents of a block \mathcal{S}_o is information that ν attributes to a partition whose only block were \mathcal{S}_o , i.e., $\nu(\omega, p_{\mathcal{S}}(\cdot|\mathcal{S}_o), \{\mathcal{S}_o\})$. With a slight abuse notation, when the partition involves a single block we simply write $\nu(\omega, p_{\mathcal{S}}(\cdot|\mathcal{S}_o), \mathcal{S}_o)$. A W -measure ν is said to be *composable* if:

$$\nu(\omega, p_{\mathcal{S}}, P) = \sum_{\mathcal{S}_o \in P} p_{\mathcal{S}}(\mathcal{S}_o) \nu(\omega, p_{\mathcal{S}}(\cdot|\mathcal{S}_o), \mathcal{S}_o)$$

Given a set \mathcal{S} of secrets, a W -measure ν is said to be *monotonic for blocks* if, for every worth assignment ω , every probability distribution $p_{\mathcal{S}}$ on \mathcal{S} , and all subsets $\mathcal{S}', \mathcal{S}''$ of \mathcal{S} s.t. $\mathcal{S}' \subseteq \mathcal{S}''$, it is the case that $\nu(\omega, p_{\mathcal{S}}(\cdot|\mathcal{S}'), \mathcal{S}') \geq \nu(\omega, p_{\mathcal{S}}(\cdot|\mathcal{S}''), \mathcal{S}'')$. In case ν quantifies uncertainty, the inequality is reversed.

3 Proposing W -measures

In this section assume that the set \mathcal{S} of secrets follows a probability distribution $p_{\mathcal{S}}$, and it is composed by fields in the set \mathcal{F} . For any $\mathcal{S}' \subseteq \mathcal{S}$ we denote by $p_{\mathcal{S}}(\cdot|\mathcal{S}')$ the normalization of $p_{\mathcal{S}}$ w.r.t. \mathcal{S}' , i.e., for every $s \in \mathcal{S}$, $p_{\mathcal{S}}(s|\mathcal{S}') = \frac{p_{\mathcal{S}}(s)}{p_{\mathcal{S}}(\mathcal{S}'})$ if $s \in \mathcal{S}'$, and $p_{\mathcal{S}}(s|\mathcal{S}') = 0$ otherwise. To illustrate our W -measures, we refer to the following general example throughout this section.

Example 7. Let $\mathcal{S} = \{000, 001, 010, 011, 100, 101, 110, 111\}$ be the set of three-bit secrets, where set of fields is $\mathcal{F} = \{f_1, f_2, f_3\}$ and $\text{domain}(f_1) = \text{domain}(f_3) = \{0, 1\}$. Assume a uniform probability distribution on secrets $p_S(s) = 0.125$ for all $s \in \mathcal{S}$, and consider the worth assignment ω in Table 2.

f	\emptyset	$\{f_1\}$	$\{f_2\}$	$\{f_3\}$	$\{f_1, f_2\}$	$\{f_1, f_3\}$	$\{f_2, f_3\}$	$\{f_1, f_2, f_3\}$
$\omega(f)$	0	2	1	0.5	3	2.5	1.5	3.5

Table 2. Worth assignment for Example 7

3.1 Deducible worth

The first W -measure we propose quantifies worth in a scenario where the adversary is allowed to guess any derived secret, but cannot tolerate any mistake in guesses. Intuitively, the worth of a block of secrets is the worth of the largest derived secret that can be inferred with certainty.

Example 8. In block of secrets $\{001, 011\}$, the first and third fields are the same. Therefore, once the adversary infers that the secret belongs to this block, the contents of the first and third fields are certain, even if the secret is not known yet. On the other hand, from block $\{010\}$ the adversary can infer the contents of all of the three fields, whereas in block $\{110, 001\}$ there is no invariant field, and thus the adversary cannot deduce the contents of any of them.

Given a probability distribution p_S on secrets, the *deducible fields* from a subset $\mathcal{S}' \subseteq \mathcal{S}$ are defined by:

$$\text{ded}(p_S, \mathcal{S}') = \{f \in \mathcal{F} \mid \forall s', s'' \in \text{supp}(\mathcal{S}') \ s'[f] = s''[f]\},$$

where $\text{supp}(\mathcal{S}')$ is the projection onto set \mathcal{S}' of the support of p_S .

To a conservative adversary concerned about the worst possible observable (i.e., block) the system could produce, the set of *deducible fields* from partition P on \mathcal{S} are the fields deducible from every block in this partition

$$\text{ded}(p_S, P) = \bigcap_{\mathcal{S}_o \in P} \text{ded}(p_S, \mathcal{S}_o).$$

The *maximum derived secret* from a block or a partition is the derived secret composed from the deducible fields of this block or partition. Note that, since the worth assignment is monotonic, the maximum worth that can be obtained with certainty from a block or partition is the worth of their maximum derived secret.

Example 9. For partition $P = \{\{000, 001, 100, 101\}, \{010, 110\}, \{011, 111\}\}$ on \mathcal{S} , the maximum derived secret is $\{f_2\}$, which is the set of common deducible fields to all of its blocks.

The *deducible worth* quantifies the worth a conservative adversary assigns to a partition.

Definition 3. The deducible worth is a WW -measure defined as:

$$WDED(\omega, p_S, P) = \omega(\text{ded}(p_S, P))$$

Note that the deducible worth is not a composable W -measure.

3.2 Probability of guessing

Assume, for simplicity, that the elements of \mathcal{S} are in decreasing order of probabilities, i.e., if $1 \leq i < j \leq |\mathcal{S}|$ then $p_S(s_i) \geq p_S(s_j)$. Then the *probability of guessing S* in n tries is defined as $PG_n(p_S) = \sum_{i=1}^n p_S(s_i)$. The *conditional probability of guessing S* in n tries given P is defined as $PG_n(p_S|P) = \sum_{\mathcal{S}_o \in P} p_S(\mathcal{S}_o) PG_n(p_S(\cdot|\mathcal{S}_o))$, where it is assumed that elements of \mathcal{S} within each distribution $p_S(\cdot|\mathcal{S}_o)$ are in decreasing order of probabilities as well. The operational interpretation of probability of guessing in n tries is: The adversary can pose questions *Is $S = s$?* for some $s \in \mathcal{S}$, and the measure quantifies the probability of guessing the whole secret in n tries, since a rational adversary would order his guesses from the most likely to the least likely one.

When worth assignment is taken into account, a more general scenario can be modeled. The attacker is still allowed to pose n questions *Is $S = s$?*, but the generalized W -measure quantifies the expected worth the adversary can extract from an attack according to some WW -measure ν .

Example 10. Consider the secrets in block $\mathcal{S}' = \{s_1, s_2, s_3, s_4, s_5\}$, with distribution $p_S(\cdot|\mathcal{S}')$. Assume the adversary has $n = 2$ questions to pose and chooses *Is $S = s_1$?* as the first. With probability $p_S(s_1|\mathcal{S}')$ the guess is correct, yielding a worth of $\omega(\mathcal{F})$ corresponding to learning the whole secret. If the first guess is wrong, the adversary picks *Is $S = s_2$?* as a second guess, and similarly, with probability $p_S(s_2|\mathcal{S}')$ will obtain worth $\omega(\mathcal{F})$. Calling $\mathcal{X} = \{s_1, s_2\}$ the set of 2 guesses chosen by the adversary, the expected worth obtained in case of success is $p_S(\mathcal{X}|\mathcal{S}')\omega(\mathcal{F})$. After 2 tries, however, the adversary has not inspected set of secrets $\bar{\mathcal{X}} = \mathcal{S}' \setminus \mathcal{X} = \{s_3, s_4, s_5\}$ with probability $p_S(\bar{\mathcal{X}}|\mathcal{S}')$. Adopting a WW -measure ν , this subset carries worth of $\nu(\omega, p_S(\cdot|\bar{\mathcal{X}}), \bar{\mathcal{X}})$. Therefore, the expected worth of block \mathcal{S}' is $p_S(\mathcal{X}|\mathcal{S}')\omega(\mathcal{F}) + p_S(\bar{\mathcal{X}}|\mathcal{S}')\nu(\omega, p_S(\cdot|\bar{\mathcal{X}}), \bar{\mathcal{X}})$. Of course, a rational adversary should maximize this value by choosing the best possible $\mathcal{X} \subseteq \mathcal{S}'$ s.t. $|\mathcal{X}| \leq n$.

The method of the above example is formalized by the *W-expected worth* measure.

Definition 4. Given a set \mathcal{S} of secrets distributed according to p_S , a worth assignment ω , an integer $n \geq 0$, and a WW -measure ν :

a) The *W-expected worth* of a block $\mathcal{S}' \subseteq \mathcal{S}$ of secrets is defined as:

$$WEXP_{n,\nu}(\omega, p_S(\cdot|\mathcal{S}'), \mathcal{S}') = \max_{\substack{\mathcal{X} \subseteq \mathcal{S}' \\ |\mathcal{X}| \leq n}} \left(\frac{p_S(\mathcal{X}|\mathcal{S}')\omega(\mathcal{F}) + p_S(\bar{\mathcal{X}}|\mathcal{S}')\nu(\omega, p_S(\cdot|\bar{\mathcal{X}}), \bar{\mathcal{X}})}{1} \right)$$

where $\bar{\mathcal{X}} = \mathcal{S}' \setminus \mathcal{X}$.

b) The W -expected worth of a partition \mathbf{P} on \mathcal{S} is a composable WW -measure defined as:

$$WEXP_{n,\nu}(\omega, p_S, \mathbf{P}) = \sum_{\mathcal{S}_o \in \mathbf{P}} p_S(\mathcal{S}_o) WEXP_{n,\nu}(\omega, p_S(\cdot|\mathcal{S}_o), \mathcal{S}_o)$$

3.3 Guessing entropy

For simplicity, assume that elements of \mathcal{S} are ordered by decreasing probabilities, i.e., if $1 \leq i < j \leq |\mathcal{S}|$ then $p_S(s_i) \geq p_S(s_j)$. Then the *guessing entropy* of S is defined as $NG(S) = \sum_{i=1}^{|\mathcal{S}|} i \cdot p_S(s_i)$. The *conditional guessing entropy* given \mathbf{P} is defined as $NG(p_S|\mathbf{P}) = \sum_{\mathcal{S}_o \in \mathbf{P}} p_S(\mathcal{S}_o) NG(p_S(\cdot|\mathcal{S}_o))$, where again elements of each $p_S(\cdot|\mathcal{S}_o)$ are in decreasing order of probabilities. The operational interpretation of guessing entropy is: The adversary poses questions $Is\ S = s?$ for some $s \in \mathcal{S}$, and the measure quantifies expected number of guesses needed to guess the entire secret.

When worth assignment is taken into account, a more general scenario can be modeled. The attacker is still allowed to pose questions $Is\ S = s?$ but instead of having to guess the secret as a whole, can fix a minimum worth $0 \leq w \leq \omega(\mathcal{F})$ to obtain according to some WW -measure ν . The generalized W -measure quantifies expected number of questions needed to obtain worth w from the attacks.

Example 11. Consider secrets in a block \mathcal{S}' distributed according to $p_S(\cdot|\mathcal{S}')$. The adversary wants to extract a worth of at least $w \leq \omega(\mathcal{F})$ according to some WW -measure ν . First he sets apart a subset $\mathcal{X} \subseteq \mathcal{S}'$ s.t. $\nu(\omega, p_S(\cdot|\mathcal{X}), \mathcal{X}) \geq w$. Then he inspects remaining subset $\bar{\mathcal{X}} = \mathcal{S}' \setminus \mathcal{X}$ by asking questions $Is\ S = s?$ until correctly guessing the secret. If successful, he gets a worth of $\omega(\mathcal{F}) > w$ corresponding to learning the whole secret. The expected number of guesses to inspect $\bar{\mathcal{X}}$ is given by $p_S(\bar{\mathcal{X}}|\mathcal{S}_o) NG(p_S(\cdot|\bar{\mathcal{X}}))$. Note that the set \mathcal{X} is left uninspected, and that its worth is at least w , so we just need to account for the extra guess needed if the secret belongs to $\bar{\mathcal{X}}$. That happens with probability $p_S(\mathcal{X}|\mathcal{S}_o)$. When that is the case, the adversary has already posed $|\bar{\mathcal{X}}|$ questions, so we need to add $p_S(\mathcal{X}|\mathcal{S}_o)(|\bar{\mathcal{X}}| + 1)$. The expected number of guesses is then $p_S(\bar{\mathcal{X}}|\mathcal{S}_o) NG(p_S(\cdot|\bar{\mathcal{X}})) + p_S(\mathcal{X}|\mathcal{S}_o)(|\bar{\mathcal{X}}| + 1)$. A rational adversary minimizes this value by choosing the best possible $\mathcal{X} \subseteq \mathcal{S}'$ s.t. $\nu(\omega, p_S(\cdot|\mathcal{X}), \mathcal{X}) \geq w$.

The method of the above example is formalized by the W -guessing entropy measure.

Definition 5. Given a set \mathcal{S} of secrets distributed according to p_S , a worth assignment ω , an real $0 \leq w \leq \omega(\mathcal{F})$, and a WW -measure ν :

a) The W -guessing entropy of a block $\mathcal{S}' \subseteq \mathcal{S}$ of secrets is defined as:

$$WNG_{w,\nu}(\omega, p_S(\cdot|\mathcal{S}'), \mathcal{S}') = \min_{\substack{\mathcal{X} \subseteq \mathcal{S}' \\ \nu(\omega, p_{\mathcal{S}'}(\cdot|\mathcal{X}), \mathcal{X}) \geq w}} \left(\begin{array}{c} p_S(\bar{\mathcal{X}}|\mathcal{S}') NG(p_S(\cdot|\bar{\mathcal{X}})) + \\ + p_S(\mathcal{X}|\mathcal{S}') (|\bar{\mathcal{X}}| + 1) \end{array} \right)$$

where $\bar{\mathcal{X}} = \mathcal{S}' \setminus \mathcal{X}$.

- b) The W -guessing entropy of a partition P on \mathcal{S} is a composable WN -measure defined as:

$$WNG_{w,\nu}(\omega, p_S, P) = \sum_{S_o \in P} p_S(S_o) WNG_{w,\nu}(\omega, p_S, S_o)$$

3.4 Shannon entropy

Given probability distribution p_S , the corresponding *Shannon entropy* is defined as $SE(p_S) = -\sum_s p_S(s) \log p_S(s)$, and the *conditional Shannon entropy* given P is defined as $SE(p_S|P) = -\sum_{S_o \in P} p_S(S_o) SE(p_S(\cdot|S_o))$. The operational interpretation of Shannon entropy is: The adversary can pose questions *Does $S \in \mathcal{S}'$?* for some $\mathcal{S}' \subseteq \mathcal{S}$, and the measure quantifies the expected minimum number of guesses needed to guess the entire secret.

When worth assignment is taken into account, a more general scenario can be modeled. The attacker is still allowed to pose questions *Does $S \in \mathcal{S}'$?* but instead of having to guess the secret as a whole, he fixes a minimum worth $0 \leq w \leq \omega(\mathcal{F})$ to obtain, according to a WW -measure ν . The generalized W -measure quantifies the expected number of questions necessary to obtain worth w from the attacks.

Example 12. In the scenario from Example 7, assume the adversary wants to achieve a worth of at least $w = 2.25$, according to deducible worth measure $WDED$. Since $f = \{f_1, f_3\}$ is the smallest derived secret s.t. $\omega(f) \geq 2.25$, the adversary only needs to discern the contents of $\{f_1, f_3\}$, independent of the contents of f_2 . Therefore, when posing a question *Does $S \in \mathcal{S}'$?* the adversary can always choose set \mathcal{S}' to exactly determine the desired derived secret. More specifically, each choice of a set \mathcal{S}' will contain exactly all secrets that share the same contents for the derived secret $\{f_1, f_3\}$. In this example the chosen questions would be: *Does $S \in \{000, 010\}$?*, *Does $S \in \{001, 011\}$?*, *Does $S \in \{100, 110\}$?*, and *Does $S \in \{101, 111\}$?*

Given a WW -measure ν , the adversary's strategy is to find a partition P_S on the space of secrets s.t. every block s in this partition gives a worth of at least w as measured by ν . The Shannon entropy of the secret given P_S provides a lower bound on the expected number of questions to achieve worth w , and naturally a rational adversary chooses the P_S that minimizes this value. This method is formalized by the W -Shannon entropy measure.

Definition 6. Given a set \mathcal{S} of secrets distributed according to p_S , a worth assignment ω , an real $0 \leq w \leq \omega(\mathcal{F})$, and a WW -measure ν :

- a) The W -Shannon entropy to achieve worth w of a block $\mathcal{S}' \subseteq \mathcal{S}$ of secrets is defined as:

$$WSE_{w,\nu}(\omega, p_S(\cdot|\mathcal{S}'), \mathcal{S}') = \min_{\substack{P_S \in LoI(\mathcal{S}) \\ \forall s \in P_S \ \nu(\omega, p_S(\cdot|s \cap \mathcal{S}'), s \cap \mathcal{S}') \geq w}} SE(p_{P_S}(\cdot|\mathcal{S}'))$$

where for every $s \in P_S$, $p_{P_S}(s|\mathcal{S}') = \sum_{s \in \mathcal{S}} p_S(s|\mathcal{S}')$.

b) The W -Shannon entropy to achieve worth w of a partition P on \mathcal{S} is a composable WN -measure defined as:

$$WSE_{w,\nu}(\omega, p_S, P) = \sum_{S_o \in P} p_S(S_o) WSE_{w,\nu}(\omega, p_S(\cdot|S_o), S_o)$$

3.5 Relation with traditional measures

Probability of guessing, guessing entropy and Shannon entropy are measures of information that ignore the worth of derived secrets. The following theorem shows that these traditional measures implicitly use the binary worth assignment ω_{bin} , which implies that no proper derived secret is deemed to be conveying relevant information.

Theorem 1. *Let \mathcal{S} be a set of secrets with probability distribution p_S , ω be a worth assignment, and P be any partition on \mathcal{S} . Then the following hold:*

$$PG_n(p_S|P) = WEXP_{n,\nu_{null}}(\omega_{bin}, p_S, P) \quad (\forall n \geq 0) \quad (1)$$

$$NG(p_S|P) = WNG_{1,WDED}(\omega_{bin}, p_S, P) \quad (2)$$

$$SE(p_S|P) = WSE_{1,WDED}(\omega_{bin}, p_S, P) \quad (3)$$

where ω_{bin} is the binary worth assignment $\omega_{bin}(f) = 1$, if $f = \mathcal{F}$, and $\omega_{bin}(f) = 0$ if $f \subset \mathcal{F}$; $WDED$ is the deducible worth measure; and ν_{null} is the null WW -measure s.t. $\nu_{null}(P) = 0$ for every partition P on \mathcal{S} .

4 Algebraic structure for W -measures

Measures of information are expected to satisfy some mathematical properties, such as non-negativity or some sort of monotonicity, according to the scenario of interest. The Lattice of Information was suggested as an underlying algebraic structure for deterministic systems [5, 17] from which common properties of measures can be derived.

The set of all partitions on a finite set \mathcal{S} forms a *complete lattice* called *Lattice of Information* (LoI) [14]. The order on the lattice elements is the *refinement order* \sqsubseteq on partitions. A partition P' *refines* a partition P , denoted by $P \sqsubseteq P'$, if every block in P is contained entirely in some block of P' . Equivalently, we can say that P' is *finer* than P . Formally, given two partitions P and P' on the same set \mathcal{S} :

$$P \sqsubseteq P' \Leftrightarrow \forall S_j \in P' \exists S_i \in P \text{ such that } S_j \subseteq S_i$$

In our model, we fix the deterministic system and let the elements in the LoI model possible executions. By controlling the low input to the system, the adversary chooses among executions, and therefore LoI serves as an algebraic representation of the partial order on the attack sequences the adversary can perform. To each attack sequence \hat{a} , there is one corresponding element $P_{\hat{a}}$ —i.e., the partition it induces—in the LoI of \mathcal{S} . Depending on attack set \mathcal{A} , however,

some element in the LoI of \mathcal{S} might not have corresponding attack sequence. The set of all executions actually corresponds to a sub-lattice of the LoI of \mathcal{S} .

Execution of an attack sequence can be seen as a path on the LoI: each attack sequence is mapped to an element in the lattice, and by performing an extra attack step the adversary may obtain a finer partition on the space of secrets, therefore moving up to a state with more information.

4.1 Defining the information leakage of attack sequences

As the adversary's state of knowledge w.r.t. the set of secrets evolves from a partition P to a finer partition P' , the leakage of information is correspondingly the difference in the W -measures of information (or uncertainty) between those two partitions.

Definition 7. Let ν be a W -measure on the elements in the LoI of a set \mathcal{S} of secrets. Let P and P' be two partitions in this LoI, such that $P \sqsubseteq P'$. Then leakage of P' w.r.t. P under the W -measure ν , given a distribution p_S on the secrets, is one of the following.

$$\mathcal{L}_\nu(\omega, p_S, P \rightarrow P') = \nu(\omega, p_S, P') - \nu(\omega, p_S, P) \quad (4)$$

$$\mathcal{L}_\nu(\omega, p_S, P \rightarrow P') = \nu(\omega, p_S, P) - \nu(\omega, p_S, P') \quad (5)$$

where (4) applies in case ν is a measure of information, and (5) applies in case ν is a measure of uncertainty.

4.2 Properties derivable from the LoI

Yasuoka and Terauchi [17] and Malacaria [5] showed that the refinement order in the LoI coincides with the order on probability of guessing, guessing entropy, and Shannon entropy. Intuitively, their result shows that these entropy definitions behave as good measures of information w.r.t. LoI. That is, the finer a partition is, the more information (or the less uncertainty) the measures attributes to it. Moreover, the theorem immediately implies that these entropy definitions always yield non-negative values for leakage. We require that W -measures proposed in Section 3 behave in a similar way, i.e., we require them to be *consistent w.r.t. LoI*, which is formalized in the following theorem.

Theorem 2. Let \mathcal{S} be a set of secrets composed by the fields in \mathcal{F} . For every two partitions P and P' in the LoI of \mathcal{S} , the following are equivalent:

$$P \sqsubseteq P' \quad (6)$$

$$\forall \omega \forall p_S \quad WDED(\omega, p_S, P) \leq WDED(\omega, p_S, P') \quad (7)$$

$$\forall n \forall \nu \forall \omega \forall p_S \quad WEXP_{n,\nu}(\omega, p_S, P) \leq WEXP_{n,\nu}(\omega, p_S, P') \quad (8)$$

$$\forall \mathbf{w} \forall \nu \forall \omega \forall p_S \quad WNG_{\mathbf{w},\nu}(\omega, p_S, P) \geq WNG_{\mathbf{w},\nu}(\omega, p_S, P') \quad (9)$$

$$\forall \mathbf{w} \forall \nu \forall \omega \forall p_S \quad WSE_{\mathbf{w},\nu}(\omega, p_S, P) \geq WSE_{\mathbf{w},\nu}(\omega, p_S, P') \quad (10)$$

where $n \geq 0$; $0 \leq \mathbf{w} \leq \omega(\mathbf{f})$; and ν ranges over all composable WW -measures that are consistent w.r.t. LoI plus the deducible worth measure $WDED$. In (9) and (10) ν is restricted to be monotonic for blocks.

5 Related work

Shannon [13] points out the independence of the information contents w.r.t. its representation, and gives the first steps in trying to understand how Shannon entropy would behave in a lattice of partitions. Köpf and Basin [15] proposed the model for deterministic systems we extended in this paper. Their goal, however, is to derive bounds automatically for attacks, rather than finding an algebraic foundation for measures they use. Malacaria [5], and also Yasuoka and Terachi [17], use Lattice of Information as an algebraic foundation for information measures. Whereas their work focuses on the comparison of already existing measures, ours proposes new measures taking into account the worth of secrets. In practical applications, Backes, Köpf and Rybalchenko [18], and Heusser and Malacaria [19] use model checkers and sat-solvers to determine the partitions induced by deterministic programs. The *g-leakage* framework [20] makes use of gain functions to measure benefits of different guesses for the secret, and generalizes min-entropy to measure the *expected gain* the adversary obtains. Their framework, however, does that implicitly, and is not suitable to deal with scenarios where bits have worth only if they can be guessed with certainty. Moreover, our work addresses generalizations of guessing entropy and Shannon entropy.

6 Conclusion

This paper introduces a model for capturing the worth of derived secrets by measures of quantitative information flow. We identify meaningful operational scenarios that cannot be accurately modeled by traditional measures of probability of guessing, guessing entropy, and Shannon entropy. We show how to use more general *W*-measures to incorporate the worth of derived secrets, and we prove that *W*-measures are consistent w.r.t. LoI.

Acknowledgements. The authors would like to thank Santosh S. Venkatesh for helpful discussions. Mário S. Alvim and Andre Scedrov are supported in part by AFOSR grant FA9550-11-1-0137. Additional support for Scedrov comes from NSF Grant CNS-0830949 and from ONR grant N00014-11-1-0555. Fred Schneider is supported in part by AFOSR grants F9550-06-0019 and FA9550-11-1-0137, National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft.

References

1. Cachin, C.: Entropy Measures and Unconditional Security in Cryptography. PhD thesis, ETH Zürich (1997) Reprint as vol. 1 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-185-7, Hartung-Gorre Verlag, Konstanz, 1997.
2. Smith, G.: On the foundations of quantitative information flow. In: FOSSACS. (2009) 288–302

3. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Quantitative notions of leakage for one-try attacks. In: Proceedings of the 25th Conf. on Mathematical Foundations of Programming Semantics. Volume 249 of Electronic Notes in Theoretical Computer Science., Elsevier B.V. (2009) 75–91
4. Massey: Guessing and entropy. In: Proceedings of the IEEE International Symposium on Information Theory, IEEE (1994) 204
5. Malacaria, P.: Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. CoRR **abs/1101.3453** (2011)
6. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. J. of Logic and Computation **18**(2) (2005) 181–199
7. Malacaria, P.: Assessing security threats of looping constructs. In Hofmann, M., Felleisen, M., eds.: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17–19, 2007, ACM (2007) 225–235
8. Malacaria, P., Chen, H.: Lagrange multipliers and maximum information leakage in different observational models. In Úlfar Erlingsson and Marco Pistoia, ed.: Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security (PLAS 2008), Tucson, AZ, USA, ACM (June 2008) 135–146
9. Moskowitz, I.S., Newman, R.E., Syverson, P.F.: Quasi-anonymous channels. In: Proc. of CNIS, IASTED (2003) 126–131
10. Moskowitz, I.S., Newman, R.E., Crepeau, D.P., Miller, A.R.: Covert channels and anonymizing networks. In Jajodia, S., Samarati, P., Syverson, P.F., eds.: Workshop on Privacy in the Electronic Society 2003, ACM (2003) 79–88
11. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Inf. and Comp. **206**(2–4) (2008) 378–401
12. Alvim, M.S., Andrés, M.E., Palamidessi, C.: Information Flow in Interactive Systems. In Gastin, P., Laroussinie, F., eds.: Proceedings of the 21th International Conference on Concurrency Theory (CONCUR 2010), Paris, France, August 31–September 3. Volume 6269 of Lecture Notes in Computer Science., Springer (2010) 102–116
13. Shannon, C.: The lattice theory of information. Information Theory, IRE Professional Group on **1**(1) (feb. 1953) 105 –107
14. Landauer, J., Redmond, T.: A lattice of information. In: Proc. Computer Security Foundations Workshop VI. (June 1993) 65 –70
15. Köpf, B., Basin, D.: Automatically deriving information-theoretic bounds for adaptive side-channel attacks. J. Comput. Secur. **19**(1) (January 2011) 1–31
16. Alvim, M.S., Scedrov, A., Schneider, F.B.: Not all bits are equal: Incorporating “worth” into information-flow measures. Technical report (2013) <http://hans.math.upenn.edu/~msalvim/papers/nabae-TechRep.pdf>.
17. Yasuoka, H., Terauchi, T.: Quantitative information flow — verification hardness and possibilities. In: Proc. 23rd IEEE Computer Security Foundations Symposium (CSF ’10). (2010) 15–27
18. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: IEEE Symposium on Security and Privacy. (2009) 141–153
19. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACSAC ’10, New York, NY, USA, ACM (2010) 261–269
20. Alvim, M.S., Chatzikokolakis, K., Palamidessi, C., Smith, G.: Measuring information leakage using generalized gain functions. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2012) 265–279

Abstract channels, gain functions and the information order

B Espinoza¹, AK McIver², LA Meinicke³, CC Morgan⁴ and G Smith¹

¹ School of Computing and Inf. Sciences, Florida International University, USA

² Dept. Computer Science, Macquarie University, Australia

³ School of Inf. Technology and Electrical Engineering, U. Queensland, Australia

⁴ School of Computer Science and Engineering, UNSW, Australia

A probabilistic channel takes secret inputs to observable outputs; given the input’s *prior* distribution, the outputs induce a *posterior* distribution whose difference from the prior- can be interpreted as the channel’s information leakage.

Generalised gain-function- or disorder tests [1, 3] can determine an information-leakage order on channels. Yet their conventional stochastic matrix presentation contains redundant information with respect to that order, i.e. with respect to leakage alone: examples are duplication, scaling or permutation of columns.

We introduce *abstract channels* by quotienting over testing equivalence; and the *Coriaceous Conjecture* [1], a completeness result, then equates the testing order to a “structural” order based on the matrices alone [2, 4]. That structural form suggests a quantitative generalisation of the Lattice of Information [1, 3].

We briefly illustrate with an example. Consider the following channels:

$$C_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1/4 & 1/2 & 1/4 \\ 1/2 & 1/3 & 1/6 \end{pmatrix} \quad C_2 = \begin{pmatrix} 2/5 & 0 & 3/5 \\ 1/10 & 3/4 & 3/20 \\ 1/5 & 1/2 & 3/10 \end{pmatrix}$$

While they may appear superficially to be very different, in fact they are semantically the *same* channel with respect to leakage—both map a prior distribution to the same distribution of posterior distributions. For if we merge the “similar” columns 2 and 3 of C_1 , and 1 and 3 of C_2 , we get the same *reduced matrix*, which can be seen as a canonical representation of an abstract channel.

Moreover C_1 and C_2 are equivalent with respect to the structural *composition refinement* preorder \sqsubseteq_\circ of [1]. Composition equivalence coincides with semantic equivalence, making \sqsubseteq_\circ a partial order on abstract channels. Also, \sqsubseteq_\circ coincides with the information leakage order, by the Coriaceous Conjecture [1, 4].

References

1. M. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. *Proc. 25th IEEE CSF* pp265–279. 2012.
2. A. McIver, L. Meinicke, and C. Morgan. Compositional closure for Bayes Risk in probabilistic noninterference. *Proc. ICALP*, LNCS 6199 pp223–35. Springer, 2010.
3. A. McIver, L. Meinicke, and C. Morgan. A Kantorovich-monadic powerdomain for information hiding, with probability and nondeterminism. *Proc. LiCS* pp461–70. IEEE 2012.
4. A. McIver, L. Meinicke, and C. Morgan. Draft proof of the Coriaceous Conjecture. <http://www.dagstuhl.de/mat/index.en.phtml?12481>

MAP-REDUCE Enforcement Framework of Information Flow Policies

Minh Ngo, Fabio Massacci, and Olga Gadyatskaya

University of Trento, Italy
{surname}@disi.unitn.it

Abstract. We propose a flexible framework that can be easily customized to enforce a large variety of information flow properties. Our framework combines the ideas of secure multi-execution and map-reduce computations. The information flow property of choice can be obtained by simply changes to a map (or reduce) program that control parallel executions.

We present the architecture of the enforcement mechanism and its customizations for non-interference (NI) (from Devriese and Piessens) and some properties proposed by Mantel, such as removal of inputs (RI) and deletion of inputs (DI), and demonstrate formally soundness and precision of enforcement for these properties.

Keywords: Runtime enforcement, information flow, secure multi-execution

1 Introduction

Information flow properties define the acceptable behaviours of computer programs with respect to allowed and forbidden flows of information. The most well-known information flow property is *non-interference* (NI), which roughly requires that the input data classified as confidential (also called secret, or high) should not influence the public (low) outputs [8, 7].

By weakening or strengthening the definition of NI, security researchers have proposed different information flow properties [15–17, 23]. For instance, the definition of NI in [8] assumes that if there is no high input, then there is no high output. Yet, this assumption does not always hold. In [17], the *generalized non-interference* (GNF) property is defined for systems that generate high outputs even if there are no high inputs.

To the best of our knowledge, there is no proposal in the literature with a unified approach to the enforcement of multiple information flow properties. The existing enforcement mechanisms (e.g. [2, 5, 7, 14, 22]) can be configured to accommodate different information flow policies that identify what is confidential and what is public, and what are the authorized flows in the security lattice [7, 20], and, sometimes, they can as well enforce declassification policies¹ (e.g. [1]).

¹ These policies are required when one needs to disclose information that depends on confidential data in some way, see e.g. [21] for details.

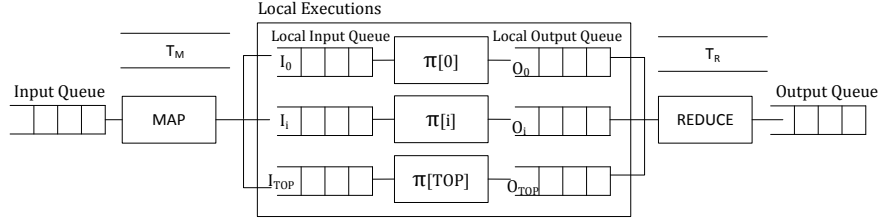


Fig. 1: Architecture of enforcement mechanisms

Yet, the adaptation of an existing enforcement mechanism (for example, for NI) to enforce another property (for instance, GNF) is not straight-forward.

We aim to fill this gap by providing an enforcement framework that can be extended by different information flow properties. The framework is inspired by the MAP-REDUCE approach from Google [12]; and generalizes the secure multi-execution (SME) technique proposed by Devriese and Piessens in [7] so that it can enforce other information flow properties, e.g. properties from [15]. The main idea is to execute multiple “local” instances of the original program, feeding different inputs to each instance of the program. The local inputs are produced from the original program inputs by the MAP component, depending on the security levels and the input channels. Upon receiving the necessary data (for instance, after each individual program instance is terminated), the REDUCE component collects the local outputs and generates the common output, thus ensuring that the overall execution is secure. MAP and REDUCE are customizable and by changing their programs the user can easily change the enforced property. Two simple tables (T_M and T_R) tell MAP and REDUCE what they should do when receiving respectively input and output requests from local executions on a channel. Table 1 summarizes the configurations for some sample properties.

The rest of the paper is organized as follows. §2 overviews the idea of our approach and the architecture of the enforcement framework; §3 introduces the semantics of controlled programs. §4 presents the formalization of the framework; §5 describes the enforcement mechanisms for the chosen information flow properties. Soundness and precision of the constructed enforcement mechanisms are postulated in §6. We discuss future extensions of the framework in §7 and its limitations in §8. Then we discuss the related work in §9 and conclude in §10.

Tab. 1: Enforcement mechanisms for the selected information flow properties

Property	Section	Components		
		MAP	REDUCE	T_M/T_R
Removal of inputs [15]	§5.1	Fig.9c	Fig.9d	Fig.9a,9b
Deletion of inputs [15]	§5.2	Fig.12c	Fig.9d	Fig.12a,12b
Termination (in)sensitive non-interference [7]	§5.3	Fig.13	Fig.9d	Fig.9a,9b

2 Overview

Fig. 1 depicts the general architecture of the enforcement mechanism for an information flow property on a program π . It is composed by a stack EX of local executions ($\pi[0], \dots, \pi[TOP]$, where TOP is the index of the top of the stack), global input and output queues, the MAP and REDUCE components, and the tables T_M and T_R .

Local executions (instances of the original program that are executed in parallel and are unaware of each other) are separated from the environment

input and output actions by the enforcement mechanism. A local execution has its own input and output queues. The local input (resp. output) queue of a local execution contains the input (resp. output) items that can be freely consumed (resp. generated) by this local execution. MAP and REDUCE are responsible for respectively the global input queue containing the input items from the external environment (received from the user or other input channels), and the global output queue containing the output items filtered by the enforcement mechanism to the environment.

When a local execution needs an input item that is not yet ready in its local input queue, it will request the help of MAP by emitting an *interrupt signal* (or just *signal* for short). When different local executions request values from the same channel, there will be only one actual input action performed by the enforcement mechanism. After the value is read, MAP will distribute it to local executions, replacing the actual value by the default (fake) one, if necessary. Similarly, when a local execution generates an output item, the output item will be handled by REDUCE.

MAP and REDUCE can also autonomously send and, respectively, collect items from local queues. For example, upon receiving an input item from the environment, MAP can send it to *all* local executions that satisfy a predicate. The parallel broadcast and parallel collection to and from local processors are the characteristic features of MAP-REDUCE programs [12]; this explains our choice for the name of the enforcement mechanism.

The actions of MAP (respectively REDUCE) on an input (output) request from a local execution depend on the configuration information in the table T_M (T_R). These components of the enforcement mechanism are customized depending on the desired information flow property. The framework components configured to implement the chosen information flow properties are listed in Tab. 1 (for each selected property the table contains pointers to the actual component configurations).

The configuration of input and output actions of local executions is based on two privileges: *ask* (a) and *tell* (t). If a local execution has the *ask* privilege on the input channel c , then MAP can fetch the input item from the environment upon receiving the interrupt signal from a local execution. If a local execution has the *tell* privilege on the input channel c , then this local execution can get the real value from the channel c when MAP broadcasts the input item to local executions, otherwise it will get a default value. If a local execution has the *ask* privilege on the output channel c , then REDUCE will actually ask the execution for the real value that it wished to send to c . Otherwise, REDUCE will just replace it with a default value. If a local execution has the *tell* privilege on the output channel c , it can invoke REDUCE to send the values generated by itself to c .

Notice that an execution may have only one privilege. For example, an execution with the *ask* but not the *tell* privilege in T_R will provide the real value to REDUCE, but will not be able to invoke REDUCE to put the value in the external output. It will have to wait for somebody else with the *tell* privilege on the channel to produce an output.

$$\begin{array}{c}
\text{INP} \quad \frac{\pi = \text{input } x \text{ from } c \quad I = \vec{v}.I' \quad \vec{v}[c] \neq \perp}{\Delta, \text{prg}:\pi, \text{mem}:m, \text{in}:I \rightarrow \Delta, \text{prg}:\text{skip}, \text{mem}:m[x \mapsto \vec{v}[c]], \text{in}:I'} \\
\text{OUTP} \quad \frac{\pi = \text{output } e \text{ to } c \quad \vec{v} = \vec{\perp}[c \mapsto m(e)]}{\Delta, \text{prg}:\pi, \text{out}:O \rightarrow \Delta, \text{prg}:\text{skip}, \text{out}:O.\vec{v}}
\end{array}$$

Fig. 3: Semantics of the input and output instructions of controlled programs

3 Semantics of Controlled Programs

Our model programming language is close to the one used in the SME paper [7]. Valid values in this language are boolean values (**T** and **F**) or non-negative integers. A program π is an instruction composed from the terms described in Fig. 2. In this figure π , e , x , and c are meta-variables for instructions, expressions, variables, and input/output channels respectively.

We model an input (output) item as a vector and define input (output) of program instances as queues. We use vectors of channel to accommodate forms in which multiple fields are submitted simultaneously but are classified differently (e.g. credit card numbers vs. user names). An *input vector* \vec{v} is a mapping from input channels to their values, $\vec{v} : C_{in} \rightarrow \Sigma \cup \{\perp\}$, where Σ is the set of all non-negative integer and boolean values, and the value \perp is the special undefined value. An *output vector* \vec{v} is a mapping from output channels to their values, $\vec{v} : C_{out} \rightarrow \Sigma \cup \{\perp\}$.

Given a vector \vec{v} and a channel c , the *value of the channel* is denoted by $\vec{v}[c]$. The symbol $\vec{\perp}$ denotes a vector mapping all channels to \perp . To simplify the formal presentation, in the sequel w.l.o.g. we assume that each input and output operation only affect one channel at a time. Thus, for each vector, there is only one channel c such that $\vec{v}[c] \neq \perp$.

Let queue Q be a sequence of elements $q_1 \dots q_n$. We denote the addition of a new element to the queue Q as $Q.q$, or $q_1 \dots q_n.q$; the removal of the first element from the queue Q is denoted by $q_2 \dots q_n$. By ϵ we denote an empty queue.

To define an execution configuration, we use a set of labelled pairs. A labelled pair is composed by a label and an object and is written in the form of *label:object*. The *label* is attached to the *object* to differentiate this object from the others, so each label occurs only once.

An *execution configuration* of a program is a set $\{\text{prg}:\pi, \text{mem}:m, \text{in}:I, \text{out}:O\}$, where π is the instruction to be executed, m is the memory (a function mapping variables to values), I (O , respectively) is the queue of input (output) vectors.

The operational semantics of the input and output instructions of the model language, as the most interesting, is described in Fig. 3. The conclusion part of each semantic rule is written as $\Delta, \Gamma \Rightarrow \Delta, \Gamma'$, where Δ denotes the elements of the execution configuration that are unchanged upon the transition. The semantics of the comma “,” in the expression Δ, Γ is the disjoint union of Δ and Γ . We abuse the notation of the memory function $m(\cdot)$ and use it to evaluate expressions to values. When an output command sends a value of e to the channel c , an output vector $\vec{v} = \vec{\perp}[c \mapsto m(e)]$ is inserted into the output queue, where \vec{v} is the vector with all undefined channels, except c that is mapped to $m(e)$, so $\vec{v}[c'] = \perp$ for all $c' \neq c$ and $\vec{v}[c] = m(e)$. For the lack of space we do not provide

$\pi ::=$	<i>instructions :</i>
$ x := e$	<i>assignment</i>
$ \pi; \pi$	<i>sequence</i>
$ \text{if } e \text{ then } \pi \text{ else } \pi$	<i>if</i>
$ \text{while } e \text{ do } \pi$	<i>while</i>
$ \text{skip}$	<i>skip</i>
$ \text{input } x \text{ from } c$	<i>input</i>
$ \text{output } e \text{ to } c$	<i>output</i>

Fig. 2: Language instructions

$$\begin{array}{l}
\text{LINP1} \frac{EX[i].st = \mathbf{E} \quad \pi = \mathbf{input} \ x \ \mathbf{from} \ c \quad dequeue(I, c) = (val, I') \quad val \neq \perp}{\Delta, EX[i].prg:\pi, EX[i].mem:m, EX[i].in:I \Rightarrow \Delta, EX[i].prg:\mathbf{skip}, EX[i].mem:m[x \mapsto val], EX[i].in:I'} \\
\\
\text{LINP2} \frac{EX[i].st = \mathbf{E} \quad \pi = \mathbf{input} \ x \ \mathbf{from} \ c \quad dequeue(I, c) = (\perp, I')}{\Delta, EX[i].stt:\mathbf{E}, EX[i].int:\perp \Rightarrow \Delta, EX[i].stt:\mathbf{S}, EX[i].int:c}
\end{array}$$

Fig. 4: Semantics of the input instruction of $\pi[i]$

the other rules; the interested reader can find the semantics of the controlled programs and the framework components in the full version of this paper [18].

An *execution* of the program π is a finite sequence of configuration transitions $\gamma_0 \rightarrow \gamma_1 \rightarrow \dots \rightarrow \gamma_k$, where $\gamma_0 = \{\text{prg}:\pi, \text{mem}:m_0, \text{in}:I, \text{out}:\epsilon\}$ is the initial configuration, m_0 is the function mapping every variable to the initial value, and k is the number of transitions. The transition sequence can be also written as $\gamma_0 \rightarrow^* \gamma$ if the exact number of transitions does not matter. The program *terminates* if there exists a configuration $\gamma_f = \{\text{prg}:\mathbf{skip}, \text{mem}:m, \text{in}:\epsilon, \text{out}:O\}$ such that $\gamma_0 \rightarrow^* \gamma_f$. We denote the whole derivation sequence by $(\pi, I) \Downarrow O$ using the big step notation.

4 Semantics of the Enforcement Mechanism

We now specify the semantics of the enforcement mechanism components: local executions, the programs of MAP and REDUCE. The general approach is that execution of parallel programs is modeled by the interleaving of concurrent atomic instructions [13] so each transition rule either by a local execution, by MAP, or by REDUCE is a step of the enforcement mechanism as a whole.

Local executions. Each local execution is associated with a unique identifier i , that is its number on the stack EX . A local execution can be in one of the two states: **E** (Executing) or **S** (Sleeping). Initially, the state of all local executions is **E**. A local execution moves from **E** to **S** when it has sent an interrupt signal to require an input item that is not ready in its local input queue, or to signal that it has generated an output item. A local execution moves from **S** to **E** when it is awoken by the MAP component (the input item it required is ready) or by the REDUCE component (its output item is consumed). The semantics for the local execution instructions is the direct adaptation of the controlled programs semantics with catering for the switch to and from the **S** state when doing inputs.

We provide the rules for input instruction in Fig. 4, where $dequeue(Q, c)$ returns the first value from the channel c in the queue Q and the rest of the queue. When the input instruction is executed and the input item required is in the local input queue, this item will be consumed (rule LINP1). Otherwise, the local execution emits an input interrupt signal c and moves to the sleep state (rule LINP2).

An *execution configuration of a local execution* $\pi[i]$ is a set $\text{LECS}_i \triangleq \{EX[i].stt:st, EX[i].int:signal, EX[i].prg:\pi, EX[i].mem:m, EX[i].in:I, EX[i].out:O\}$, where EX is the global stack of local execution, i denotes the i -th execution, st is the state of the local execution, $signal$ is the interrupt signal sent by the local execution, π is an instruction to be executed, m is the memory, and I and O are queues of input and output vectors respectively.

$$\begin{array}{c}
\text{MAP} \frac{\pi_M = \mathbf{map}(e, c, PRED[]) \quad S = \{i \in \{0, \dots, TOP\} : PRED[i]\} \\
\text{LECS} = \bigcup_{i \in S} \{EX[i].in:I\} \quad \vec{v} = \perp[c \mapsto m(e)] \quad \text{LECS}' = \bigcup_{i \in S} \{EX[i].in:I.\vec{v}\}}{\Delta, \text{map:prg}:\pi_M, \text{LECS} \Rightarrow \Delta, \text{map:prg:skip}, \text{LECS}'} \\
\\
\text{CLNE} \frac{\pi_M = \mathbf{clone}(PRED[], PRIV_{T_M}, PRIV_{T_R}) \quad S = \{i \in \{0, \dots, TOP\} : PRED[i]\} \\
\text{LECS} = \bigcup_i \text{LECS}_i \quad \text{LECS}' = \text{LECS} \cup \bigcup_{i \in S} \text{fork}(\text{LECS}_i, TOP + \text{assignIndex}(i)) \\
TOP' = TOP + |S| \quad (T'_M, T'_R) = \text{assign}(T_M, T_R, TOP, TOP', PRIV_{T_M}, PRIV_{T_R})}{\Delta, \text{tm}:T_M, \text{tr}:T_R, \text{top}:TOP, \text{map:prg}:\pi_M, \text{LECS} \Rightarrow \Delta, \text{tm}:T'_M, \text{tr}:T'_R, \text{top}:TOP', \text{map:prg:skip}, \text{LECS}'}
\end{array}$$

Fig. 6: Semantics of the MAP instructions **map** and **clone**

MAP. A MAP program is normally composed of three steps: the input retrieval step, the value distribution step and the wake up step. In the first step, an input item is fetched by performing an actual input action from the specified channel, or by using the default value (val_{def}). In the second step, a real input item or the default item is sent to local executions. These two steps depend on the configuration in T_M . In the third step, local executions are awoken if a certain condition is satisfied, e.g., these local executions were waiting for input items and they have received the input items they required.

In addition to the instructions in Fig. 2 (except for the output instruction replaced by the map instruction), π_M may also contain the instructions described in Fig. 5, where $PRED[] \triangleq \lambda x. Pred(x)$ is a meta-variable for predicates. Evaluation of the predicate $PRED[]$ on the configuration of the local execution $\pi[i]$ is denoted as $PRED[i]$.

$\pi_M ::= \dots$	<i>instructions :</i>
$\mathbf{map}(e, c, PRED[])$	<i>map</i>
$\mathbf{wake}(PRED[])$	<i>wake</i>
$\mathbf{clone}(PRED[], PRIV_{T_M}, PRIV_{T_R})$	<i>clone</i>

Fig. 5: The MAP instructions

The execution of *map*, *wake*, or *clone* instruction is applied simultaneously to all local executions $\pi[i]$ such that $PRED[i]$ is true as follows. The execution of the map instruction sends the value of the expression e to the input queues of all local executions. The value sent is considered as a value from the channel c . The execution of the wake instruction wakes all local executions $\pi[i]$ up and removes the interrupt signals generated by those local executions (if there were some). The execution of the clone instruction clones the configuration of each local execution $\pi[i]$. The new executions will be appended to the local executions stack. The state of the new executions is **S**. The privileges of the new local executions are copied from the lists of privileges $PRIV_{T_M}$ and $PRIV_{T_R}$. $PRIV_{T_M}$ ($PRIV_{T_R}$, respectively) is an input (output, respectively) privilege configuration template which varies depending on the enforced property. We give an example of such templates in §5.2, where the enforced property requires cloning.

A *configuration of the MAP component* is a set $\{\text{map:prg}:\pi_M, \text{map:mem}:m\}$, where π_M is the instruction to be executed, and m is the memory.

The semantics of instructions of assignment, sequence, if, while, and skip of MAP is quite similar to the semantics of corresponding instructions of controlled programs. The output instruction is not used in π_M . The semantics of the two most interesting instructions map and clone is described in Fig. 6. For the map, wake, and clone instructions, if there is no i such that $PRED[i]$ holds, then the execution of these instructions makes all local executions to move from their current configurations to themselves.

$$\begin{array}{c}
\text{OUTR} \frac{\pi_R = \mathbf{output} \ e \ \mathbf{to} \ c \quad \text{red.mem} = m \quad \vec{v} = \perp[c \mapsto m(e)]}{\Delta, \text{red:prg}:\pi_R, \text{out}:O \Rightarrow \Delta, \text{red:prg}:\mathbf{skip}, \text{out}:O.\vec{v}} \\
\\
\text{WAKR} \frac{\text{LECS} = \bigcup_{i \in S} \{\pi_R = \mathbf{wake}(PRED[]), EX[i].\text{int:signal}, EX[i].\text{stt:S}\} \quad S = \{i \in \{0, \dots, TOP\} : PRED[i]\} \quad \text{LECS}' = \bigcup_{i \in S} \{EX[i].\text{int}:\perp, EX[i].\text{stt:E}\}}{\Delta, \text{red:prg}:\pi_R, \text{LECS} \Rightarrow \Delta, \text{red:prg}:\mathbf{skip}, \text{LECS}'}
\end{array}$$

Fig. 8: Semantics of the REDUCE instructions **output** and **wake**

The bijective function $assignIndex : S \rightarrow \{1, \dots, |S|\}$ assigns and returns a unique index of the element i in the set S (the index starts from 1). The function $fork(\text{LECS}_i, j)$ makes a copy of the local execution $\pi[i]$; the new execution can be referred as $EX[j]$. The function $assign(T_M, T_R, TOP, TOP', PRIV_{T_M}, PRIV_{T_R})$ modifies tables T_M and T_R by adding new columns for the newly cloned processes and the corresponding values for the privileges from $PRIV_{T_M}$ and $PRIV_{T_R}$ for the input and output channels for these processes.

REDUCE. The REDUCE component controls the output actually generated by the enforcement mechanism. A REDUCE program π_R can ask an item from a local execution, send an item to the external output, clean local output queues of local executions and wake local executions up.

Except for the input instruction, that is replaced by the retrieve instruction, in addition to the instructions in Fig. 3 and the wake instruction, the REDUCE program may contain instructions described in Fig. 7. The execution of the retrieve instruction reads the value from the output queue of $\pi[i]$ and stores it into x . The execution of the clean instruction is applied to all local executions $\pi[i]$ such that $PRED[i]$ is true. This instruction removes the first vector \vec{v} of the output queue O of $\pi[i]$, where the value of $\vec{v}[c]$ is different from \perp .

A *configuration of the REDUCE component* is a set $\{\text{red:prg}:\pi_R, \text{red:mem}:m\}$, where π_R is the instruction to be executed, and m is the memory.

$\pi_R ::= \dots$ *instructions :*
 $\quad | \mathbf{retrieve} \ x \ \mathbf{from} \ (i, c) \quad \text{retrieve}$
 $\quad | \mathbf{clean}(c, PRED[]) \quad \text{clean}$

The semantics of the output and wake instructions, as the most interesting ones, is described in Fig. 8.

Fig. 7: The REDUCE instructions

The enforcement mechanism. A *configuration of an enforcement mechanism* is a set $\{\mathbf{t}_m:T_M, \mathbf{t}_r:T_R, \mathbf{top}:TOP, \mathbf{map}:M, \mathbf{red}:R, \mathbf{in}:I, \mathbf{out}:O, \bigcup_i \text{LECS}_i\}$, where T_M and T_R are configuration tables for respectively MAP and REDUCE, TOP is the index of the top of the stack of configurations of local executions EX , M and R are configurations of respectively MAP and REDUCE components, I and O are respectively the input and output queues of the enforcement mechanism, and LECS_i is the configuration of the i -th local execution.

The program of MAP (REDUCE respectively) is activated only when the previous execution of MAP (REDUCE) terminated, there is an interrupt signal c from the local execution $\pi[i]$, the state of this local execution is sleeping (S), and the instruction to be executed is an input (output) instruction. The activation of the MAP program or the REDUCE program on a signal on channel c from $\pi[i]$ will remove the signal from $\pi[i]$.

We denote the enforcement mechanism on π by $\text{EM}(\pi)$. For the initial configuration, all local input and output queues will be empty, all local executions will be in the executing state, and skip is the only instruction in MAP and REDUCE programs. The enforcement mechanism terminates when all local executions, MAP and REDUCE programs are terminated, and the global input queue is consumed completely.

The enforcement mechanism *terminates* if there exists a configuration $\gamma_f = \{\mathbf{t}_m:T_M, \mathbf{t}_r:T_R, \mathbf{top}:TOP, \mathbf{map}:M, \mathbf{red}:R, \mathbf{in}:\epsilon, \mathbf{out}:O, \bigcup_i \text{LECS}_i\}$ such that $\gamma_0 \rightarrow^* \gamma_f$, where $EX[i].\mathbf{prg}:\mathbf{skip}$ for all i , $\mathbf{map}.\mathbf{prg}:\mathbf{skip}$, and $\mathbf{red}.\mathbf{prg}:\mathbf{skip}$. We denote this whole derivation sequence by $(\text{EM}(\pi), I) \Downarrow O$ using the big step notation.

5 Configurations for the Selected Properties

In [15], Mantel proposes a uniform framework to define possibilistic information flow properties and he proves that existing possibilistic information flow properties can be expressed as a predefined basic security predicate (BSP) or conjunction of these BSPs. A BSP is generally defined in the framework of Mantel based on removal of some high inputs and events.

In the next sections, we will demonstrate configurations of our framework for enforcement of two BSPs, RI and DI, and the SME-style NI. It might not be obvious whether these properties are actually different in our model. We resolve possible doubts of the attentive reader in [18].

Let $COND[] \triangleq \lambda \vec{v}.Cond()$ be a predicate and $COND[\vec{v}]$ be the result of the evaluation of $COND[]$ on \vec{v} . We define the restriction operator on the queue Q with $COND[], Q|_{COND[]}$, that returns all \vec{v} in Q such that $COND[\vec{v}]$ is true. We will use the notation $Q|_l$, the restriction on security level l , if $Cond(l) \triangleq \lambda \vec{v}.\exists c : \vec{v}[c] \neq \perp \wedge LVL[c] = l$; and the notation $Q|_c$, the restriction on channel c , if $Cond(c) \triangleq \lambda \vec{v}.\vec{v}[c] \neq \perp$.

5.1 Removal of Inputs

The *removal of inputs* (RI) property [15] requires that if a possible trace is perturbed by removing all high input items, then the result can be corrected into a possible trace. In our notation if all high input items are replaced by the default values or removed, the input queue can be sanitized so that the program will terminate when executing on this input and the generated output will be equivalent at the low level to the original output.

Definition 1. A program π satisfies the property of removal of inputs iff for any potential value chosen as a default value,

$$\begin{aligned} \forall I : (\pi, I) \Downarrow O \implies \exists I' : I'|_L = I|_L \wedge I'|_H = (\vec{df})^* \wedge \\ \wedge \forall c \in C_{in}, \| I'|_c \| \leq \| I|_c \| \wedge (\pi, I') \Downarrow O' \wedge O'|_L = O|_L, \end{aligned}$$

where \vec{df} is a vector containing the default value, and $\| Q \|$ is the length of Q .

The enforcement mechanism of the RI property on the program π only needs two parallel programs: the high ($\pi[0]$) and the low ($\pi[1]$). We specify the full configuration of the local executions in Fig. 9. The high execution can receive (real) input values from L and H channels, while the low execution can receive only (real) input values from L channels. The high execution can write output values only to H channels, the low execution can write values only to L channels. If the interrupt signal is from $\pi[1]$, or the interrupt signal is from $\pi[0]$ and the level of channel c is H , then the input action will be performed. Otherwise, the local execution keeps sleeping.

The MAP program is described in Fig. 9c. The function $canTell(c)$ indicates whether the local execution $\pi[x]$ can receive real values from MAP: $canTell(c) \triangleq \lambda x. t \in T_M[x][c]$. If a local execution that is sleeping and waiting for an input item from a channel has received the input item required, this local execution is ready to be awoken: $isReady(c) \triangleq \lambda x. EX[x].stt = \mathbf{S} \wedge EX[x].prg = \mathbf{input\ y\ from\ } c; \pi \backslash EX[x].in = I \wedge dequeue(I, c) = (val, I') \wedge val \neq \perp$.

When there is an interrupt signal c from $\pi[i]$ on an output instruction, the REDUCE program provided in Fig. 9d is activated. If the local execution $\pi[i]$ can send items to the c channel, the output action is performed. Otherwise, there is no output action. After that the output queue of $\pi[i]$ is cleaned and only $\pi[i]$ is waken. Since the execution of the wake instruction wakes up only $\pi[i]$, the function $identical()$ is defined as $identical(i) \triangleq \lambda x. x = i$.

Example. We illustrate the enforcement mechanism of RI with the program in Fig. 10. The program has two high input channels $cH1$, $cH2$, and one high output channel $cH3$. It is not secure: with the execution of instructions at lines 3, 4, 7, and 8, the secret values from $cH1$ (line 1) and $cH2$ (line 8) can influence the value sent to the low output channel $cL3$ (line 10). In addition, the sequences of high input items are affected by the low input (line 7 and 8); for example, if the value of $l1$ is \mathbf{T} , an input item from $cH2$ will be consumed. We consider the execution of the program with the input sequence ($cH1 = \mathbf{T}$) ($cL1 = \mathbf{F}$) ($cL2 = m$) ($cH2 = M$).

The high execution in our framework executes the instructions at lines 1, 2, 3, 5, 6, 7, 9, and 10. The output generated at line 10 is ignored by REDUCE. The low execution executes the instructions from line 1 to 10. MAP reads an

	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	a
$LVL[c] = L$	t	at

(a) T_M for RI

	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	$-$
$LVL[c] = L$	$-$	at

(b) T_R for RI

```

1: if  $a \in T_M[i][c]$  then
2:   input  $x$  from  $c$ 
3:   map( $x, c, canTell(c)$ )
4:   map( $val_{def}, c, \neg canTell(c)$ )
5:   wake( $isReady(c)$ )
6: else
7:   skip

```

(c) MAP for RI for an input from c from $\pi[i]$

```

1:  $x := val_{def}$ 
2: if  $a \in T_R[i][c]$  then
3:   retrieve  $x$  from  $(i, c)$ 
4: if  $t \in T_R[i][c]$  then
5:   output  $x$  to  $c$ 
6: clean( $c, identical(i)$ )
7: wake( $identical(i)$ )

```

(d) REDUCE for RI for an output to c from $\pi[i]$

Fig. 9: Configuration of the enforcement mechanism for RI

```

1 input h1 from cH1
2 input l1 from cL1
3 if !h1 then
4   l1 := !l1
5 input l2 from cL2
6 h2 := 0
7 if l1 then
8   input h2 from cH2
9 output l2 + h2 to cH3
10 output l2 + h2 to cL3

```

Fig. 10: Running Example Program

input from cH3 for the input instruction at line 8. The output generated by the output instruction at line 9 is ignored by REDUCE.

We describe the global input, output queues, and local input, output queues in Fig. 11. The values sent to cH3 and cL3 are respectively m and $*+m$. Each column in the tables corresponds to an input/output operation. Input and output tables should be read from left to right; columns describe the input/output to each channel at time $t = 0, t = 1$, etc.

5.2 Deletion of Inputs

The property of deletion of inputs (DI) [15] requires that if we perturb a possible trace t (where $t = \beta.e.\alpha$ and there is no high input event in α) by deleting the high input event e , then the result can be corrected into a possible trace t' ($t' = \beta'.\alpha'$).

The parts β and β' are equivalent on the low input events and the high input events. In other words, the low input events and the high input events in β and β' must be the same. The parts α and α' are also equivalent on the low events and the high input events. Since there is no high input events in α , there is also no high input events in α' .

In our notation, if we have an input queue $I = I_1.\vec{v}.I_2$, where \vec{v} contains a value from a high channel and in I_2 there are either no high input items or only high input items with default values, then this input queue can be changed by replacing \vec{v} by the default vector. The obtained input queue can be sanitized by removing existing default high input items in I_2 or adding other default high input items to I_2 . The sanitized queue can be consumed completely by a clone of the original program and the output should still be equivalent at the low level to the original output generated with the input I .

Definition 2. A program π satisfies the property of deletion of inputs DI iff for any potential value chosen as a default value,

$$\begin{aligned} \forall I : I = I_1.\vec{v}.I_2 \wedge LVL[c] = H \wedge I_2|_H = (\vec{df})^* \wedge (\pi, I) \Downarrow O &\implies \\ \exists I' : I' = I'_1.I'_2 \wedge I'|_L = I|_L \wedge I'_2|_H = (\vec{df})^* \wedge (\pi, I') \Downarrow O' \wedge O'|_L = O|_L, \end{aligned}$$

Input to MAP:

	0	1	2	3
ch1	T	⊥	⊥	⊥
ch2	⊥	⊥	⊥	M
cl1	⊥	F	⊥	⊥
cl2	⊥	⊥	m	⊥

Output by REDUCE:

	0	1	2	3	4	5
ch3	⊥	⊥	⊥	⊥	m	⊥
cl3	⊥	⊥	⊥	⊥	⊥	*+m

Local Executions:

The high execution $\pi[0]$:

The local input:

ch1	T	⊥	⊥	⊥
ch2	⊥	⊥	⊥	M
cl1	⊥	F	⊥	⊥
cl2	⊥	⊥	m	⊥

The local output:

ch3	⊥	⊥	⊥	⊥	m	⊥
cl3	⊥	⊥	⊥	⊥	⊥	m

The low execution $\pi[1]$:

The local input:

ch1	F	⊥	⊥	⊥
ch2	⊥	⊥	⊥	*
cl1	⊥	F	⊥	⊥
cl2	⊥	⊥	m	⊥

The local output:

ch3	⊥	⊥	⊥	⊥	*+m	⊥
cl3	⊥	⊥	⊥	⊥	⊥	*+m

Fig. 11: Example of input and output queues for RI

	$\pi[0]$	$\pi[1]$	$\pi[i] > 1$
$LVL[c] = H$	at	a	a
$LVL[c] = L$	t	at	t

(a) T_M for DI

	$\pi[0]$	$\pi[1]$	$\pi[i] > 1$
$LVL[c] = H$	at	—	—
$LVL[c] = L$	—	at	—

(b) T_R for DI

```

1: if  $LVL[c] == H$  and  $i == 0$  then
2:   clone(identical(i),  $PRIV_{T_M}$ ,  $PRIV_{T_R}$ )
3: if  $a \in T_M[i][c]$  then
4:   input  $x$  from  $c$ 
5:   map( $x, c$ ,  $canTell(c)$ )
6:   map( $val_{def}, c$ ,  $\neg canTell(c)$ )
7:   wake(isReady(c))
8: else
9:   if  $t \notin T_M[i][c]$  then
10:    map( $val_{def}, c$ , identical(i))
11:    wake(identical(i))
12:   else
13:     skip

```

(c) MAP for DI for an input from c from $\pi[i]$

Fig. 12: Configuration of the enforcement mechanism for DI. REDUCE is in Fig. 9d.

where $\vec{v}[c] \neq \perp$ and $\vec{d\bar{f}}$ is a vector containing the default value.

DI is enforced with the idea that whenever the high execution requests a high input item, this execution will be cloned and the clone cannot receive real values from high channels. The enforcement mechanism for DI (presented in Fig. 12, REDUCE is presented in Fig. 9d) requires more than two local executions. Only the high execution $\pi[0]$ can ask for and get the high input items, other local executions will only use the default values. When the high execution is cloned the new execution is inserted into the stack of local executions. The configuration of the clones for input (respectively, output) is presented in Fig. 12a (respectively, 12b) in the column $\pi[i] > 1$; this is the privilege configuration template $PRIV_{T_M}$ ($PRIV_{T_R}$, respectively). In addition, only the low execution $\pi[1]$ can ask for low input items and generate low output items; other local executions will reuse the low input items retrieved by the low execution.

5.3 Non-Interference

The enforcement mechanism configured in this section mimics the SME-style enforcement of non-interference [7] from Devriese and Piessens, and therefore inherits also the limitations of SME formal guarantees.

Informally, a program satisfies the termination-insensitive non-interference (TINI) property if given two arbitrary inputs that are equivalent at the low level and the executions of the program on these two inputs are terminated, then the outputs generated are indistinguishable to the users at the low level. In other words, the high input items in these two inputs have no effect on what observable is to users at the low level. Termination-sensitive non-interference (TSNI) additionally requires that the secret input items do not influence the termination of the program [2].

Definition 3. A program π satisfies the property of termination-insensitive non-interference, denoted as $\pi \models TINI$, with respect to the given semantics iff

$$\forall I, I' : I|_L = I'_L \implies O|_L = O'_L,$$

where $(\pi, I) \Downarrow O$ and $(\pi, I') \Downarrow O'$.

Definition 4. A program π satisfies the property of termination-sensitive non-interference, denoted as $\pi \models TSNI$, with respect to the given semantics iff

$$\forall I, I' : I|_L = I'_L \wedge (\pi, I) \Downarrow O \implies (\pi, I') \Downarrow O' \wedge O'|_L = O|_L.$$

To implement the SME approach [7], we use the following configuration. The high execution $\pi[0]$ can only ask high input items, while for low input items it needs to wait for the values ask by the low execution $\pi[1]$. The low execution $\pi[1]$ can ask and consume only low items. If the low execution requires a high input item, the default value will be used.

The configuration tables T_M and T_R and the program for REDUCE to enforce the SME-style NI are presented in Fig. 9. However, the program for MAP is different, as shown in Fig. 13. The functions $canTell(i)$, $isReady()$ and $identical(i)$ are defined in Sec. 5.1.

```

1: if  $a \in T_M[i][c]$  then
2:   input  $x$  from  $c$ 
3:   map( $x, c, canTell(c)$ )
4:   map( $val_{def}, c, \neg canTell(c)$ )
5:   wake( $isReady(c)$ )
6: else
7:   if  $t \notin T_M[i][c]$  then
8:     map( $val_{def}, c, identical(i)$ )
9:     wake( $identical(i)$ )
10:  else
11:    skip

```

Fig. 13: MAP for SME for input from c requested from $\pi[i]$

6 Formal Properties

We formalize the soundness and precision properties of an enforcement mechanism and prove the theorems on the security guarantees that the shown enforcement mechanisms ensure with respect to the corresponding properties. Table 1 summarizes the properties and enforcement mechanisms.

Definition 5. *An enforcement mechanism is sound with respect to a property P if for all programs π the enforcement mechanism executed on π satisfies P .*

Theorem 1. *Each enforcement mechanism in Tab. 1 is sound with respect to the corresponding property, except for TSNI.*

Proof sketch: the low input items consumed and the low output items generated by the enforcement mechanism are always produced by the low execution; the high output items are always generated by the high execution. The differences among different properties come from the constraints on high input items. By using the induction technique on the length of the derivation sequence of the enforcement mechanism, we can prove that the high input items consumed by the enforcement mechanism satisfy the constraints of the enforced property. \square

The notion of precision for enforcement of a property is taken from [7, 9]. The intuition is that the enforcement mechanism does not change the visible behavior of a program that is already secure with respect to the chosen property (and in particular each I/O on specific channels). Devriese and Piessens separated by construction the input queues of each channel. Since in our formulation the channels are merged into a global stream, our definition of precision must make explicit that the partial order of input items on a channel is preserved. This observation applies also to the order of output items in output queues. In our framework, the local executions are executed in parallel with no specific order. Therefore, the total order of input items consumed by the enforcement mechanism can be different from the total order of input items in the input queue consumed by the controlled program that already obeys the desired property. However, the partial order of input items on a channel is preserved. This observation applies also to the order of output items in output queues.

Definition 6. *An enforcement mechanism is precise with respect to a property, if for any program π that satisfies the property, and for every input I , where $(\pi, I) \Downarrow O$, the actually consumed input I^* and the actual output O^* of the enforcement mechanism regardless of the order of executing local executions will be such that $I^*|_c = I|_c$ and $O^*|_c = O|_c$ for every channel c , and $(EM(\pi), I^*) \Downarrow O^*$.*

Theorem 2. *Each enforcement mechanism in Tab. 1 is precise with respect to the corresponding property, except for TINI.*

Proof sketch: let π be a program satisfying the enforced property and $(\pi, I) \Downarrow O$. Regardless of the order of executing local executions, if the low execution consumes the same low input items as in I and the high execution consumes high input items as in I , then the input consumed by the enforcement mechanism is I^* , where $I|_c = I^*|_c$ for all c (if not, then contradictions will occur). \square

We prove Th. 1 and Th. 2 in the full version of this paper [18].

7 Further Properties

Our framework can capture other properties. Other BSPs from [15] can also be enforced. Removal of events (RE) requires that if there is no high input, there is no high output. To enforce RE, when receiving an output request for a high channel from the high execution, REDUCE needs to check whether there are any other high input items different from the default values and affecting the output generated by the high execution. Enforcement of strict removal of inputs (SRI) is similar to the enforcement of RI, but only the low execution can generate output items for both high and low channels. Strict deletion of inputs, deletion of events, and backward strict deletion can be enforced by using the clone instruction and the REDUCE check mentioned above.

By modifying the privileges of local executions we can enforce new properties. A possible configuration is shown in Fig. 14, where the low execution needs to wait for high input items requested by the high execution even though the low execution can only consume default values. This option leads to a novel strict property, which we have called *substitution-deletion of inputs* (SubDI). The configuration of T_R also leads to discovery of new properties. For the lack of space we provide examples of SubDI and the properties by the modification of T_R in the full version of this paper [18].

	$\pi[0]$	$\pi[1]$
$LVL[c] = H$	at	$-$
$LVL[c] = L$	t	at

$\pi[1]$ waits for high input events from $\pi[0]$

Fig. 14: SubDI

Our framework can be extended to accommodate information flow policies represented as a complete lattice [6]. For a complete lattice with n elements, the enforcement mechanisms of RI and NI require n local executions, one for each element of the lattice. The enforcement mechanism of DI requires n local executions at initialization; it will spawn a new local execution every time a local execution at the level l (where l is not the level at the bottom of the lattice) requests an input item at the level l .

8 Limitations

Currently, the enforcement mechanism is not independent from the choice of the default values (val_{def}). We prove soundness and precision of enforcement with respect to all possible choices of the default values, and we assume that for each channel it is possible to determine a suitable (“non-leaking”) default value.

Our mechanism in §5.3 inherits the limitations of SME [7]. SME can soundly enforce TINI, but not TSNI because the low execution may terminate while the high execution may not terminate, and thus the whole enforcement mechanism does not terminate. SME (and our enforcement mechanism for NI) can precisely enforce TSNI, but not TINI.

In [11] Kashyap, Wiedermann and Hardekopf evaluate the security guarantees of SME for the termination covert channel; they have proposed to mediate the security problems of SME related to this channel with more sophisticated schedulers. In our approach we do not schedule the order of local executions, therefore, we cannot immediately adapt their suggestions. However, our framework can be extended to control the order of executing local executions by specifying a new rule to control the start of local executions, and the predicate *isReady()* used in the wake instruction.

We see one of the main limitations of our current proposal in the absence of a practical implementation. It is still an open question, whether the memory and performance overhead will be acceptable, especially for complex properties, such as DI. In addition, the fact that MAP and REDUCE are responsible for all respectively input and output operations may have influence on the performance of the enforcement mechanism. Devriese and Piessens in the original SME paper [7], as well as Bielova et al. in [4] and De Groef et al. in [9] report on complications while instrumenting SME for real browsers, which we will have to address. A working implementation is our next target.

9 Related Work

The information flow policies enforcement is a deeply investigated field. We will briefly recall the developed approaches for information flow policies enforcement and discuss the most relevant techniques in more details.

Static analysis techniques for information flow security inspect the program code in order to check whether there is any unwanted information flow. We refer the interested reader to the survey by Sabelfeld and Myers [20] with an excellent overview of static language-based approaches for information flow security.

In contrast to the static verification techniques, dynamic analysis for information flow enforcement tracks propagation of confidential information when a program is executed; an extensive review on the dynamic approach can be found in [14]. The trade-offs between static and dynamic analysis approaches are evaluated by Russo and Sabelfeld in [19].

Our choice of the multi-execution approach, despite its performance overhead which is negligible with multicore, was dictated by its advantages over the static and dynamic information flow analysis techniques. Static analysis can fall short in scenarios when the program can be composed dynamically (e.g. JavaScript); dynamic runtime monitoring can suffer from impossibility to account for the branch of execution that was not taken, and can leak control flow details [20] through the halting behaviour of the program.

Secure multi-execution [7] has inspired many researchers to push further investigation of this technique. Jaskelioff and Russo in [10] describe their adaptation of SME to Haskell and provide an SME implementation in a handy library. SME is applied to a reactive model of a browser in [4], and is implemented as a fully functional web browser FlowFox that embeds an SME-based runtime enforcement mechanism in [9]. FlowFox is a modification of Firefox, it introduces a noticeable memory and performance overhead, but works with most of the existing web sites. We plan to learn from [9] how to implement a fully working solution and how to evaluate the usability.

Barthe et al. [3] provides sound and precise enforcement of non interference through static program transformation instead of modifying the runtime environment. The transformation technique is based on the main SME idea: a program is transformed into the sequential composition of the same code, first at the low level and then at the high level. The high instance reuses the inputs of the low instance through global input buffers.

Instead of having multiple different executions, in [1] non-interference is achieved by using faceted values (pairs of two values containing low and high information). This allows to simulate multiple executions on different security levels while in fact running a single-process execution. The authors also introduce enforcement of declassification policies with their technique.

Capizzi et al. in [5] propose a shadow execution technique that is similar to SME. Shadow execution consists of replacing the original program with two copies. The private copy (the high execution) receives the confidential data, but is prevented from accessing the network. The public copy (the low execution) receives fake data, but can access the network; the results from the network are supplied also to the private copy. In this way the private copy can avail any network related functionality without leaking the confidential data.

10 Conclusion

We have presented the architecture of an extensible framework for enforcement of information flow properties. To the best of our knowledge, this is the first enforcement mechanism capable to accommodate more than one property. The main idea behind our approach is to run several local instances of a program in parallel, as in secure multi-execution [7], and carefully orchestrate processing of input and output operations of the enforced program through two components (MAP and REDUCE), and two tables (T_M and T_R).

To support our claims on the extensibility of the framework we have provided a set of configurations of the enforcement framework for enforcement of non-interference and several properties from the framework of Mantel [15]. The framework components programs for each of these properties are simple and easy to write. As for software, the correctness proof maybe complicated but writing programs that work should be easy.

Our next steps include the investigation the patterns of T_M , T_R , π_M , π_R and the property to be enforced; a proof-of-concept implementation (we have chosen to implement our framework for a web browser; however, our approach can be suitable to any platform), and extension of the framework with more properties and options for declassification.

Acknowledgements We thank the anonymous reviewers of FCS’2013 for their feedback and suggestions which greatly helped to improve the paper. This work is partly supported by the projects EU-IST-NOE-NESSOS and EU-IST-IP-ANIKETOS.

References

1. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. *SIGPLAN Not.*, 47(1):165–178, Jan. 2012.
2. G. Barthe, P. R. D’Argenio, and T. Rezk. Secure information flow by self-composition. *Math. Structures in Computer Science*, 21(6):1207–1252, 2011.

3. G. Barthe, et al. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems*, volume 7273 of *LNCS*, pages 186–202, 2012.
4. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. of NSS 2011*, pages 97–104, 2011.
5. R. Capizzi, et al. Preventing information leaks through shadow executions. In *Proc. of ACSAC 2008*, pages 322–331, 2008.
6. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
7. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of IEEE S&P 2010*, pages 109–124, 2010.
8. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of IEEE S&P 1982*, pages 11–20, 1982.
9. W. D. Groef, et al. Flowfox: a web browser with flexible and precise information flow control. In *Proc. of CCS 2012*, pages 748–759, 2012.
10. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Perspectives of Systems Informatics*, volume 7162 of *LNCS*, pages 170–178, 2012.
11. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE S&P 2011*, pages 413–428, 2011.
12. R. Lämmel. Google’s MapReduce programming model - revisited. *Sci. Comput. Program.*, 68:208–237, October 2007.
13. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.
14. G. Le Guernic. *Confidentiality enforcement using dynamic information flow analyses*. PhD thesis, Kansas State University, Manhattan, KS, USA, 2007.
15. H. Mantel. Possibilistic definitions of security - an assembly kit. In *Proc. of CSFW 2000*, pages 185–199. IEEE Computer Society, 2000.
16. D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. of IEEE S&P 1987*, pages 161–166, 1987.
17. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of IEEE S&P 1994*, pages 79–93, May 1994.
18. M. Ngo, F. Massacci, and O. Gadyatskaya. MAP-REDUCE runtime enforcement of information flow policies. Available as the Arxiv report 1305.2136. <http://arxiv.org/abs/1305.2136>. Technical Report DISI-13-019, University of Trento, 2013.
19. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of CSF 2010*, pages 186–199, july 2010.
20. A. Sabelfeld and A. Myers. Language-based information-flow security. *J. on Selected Areas in Communications*, 21(1):5–19, 2003.
21. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. on Comput. Secur.*, 17(5):517–548, Oct. 2009.
22. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.
23. A. Zakinthinos and E. Lee. A general theory of security properties. In *Proc. of IEEE S&P 1997*, pages 94–102, May 1997.

A Framework for Composing Security-typed Languages

Andreas Gampe and Jeffery von Ronne

The University of Texas at San Antonio
`{agampe,vonronne}@cs.utsa.edu`

Abstract. Users often run software programs that have access to private information that they do not wish to be disclosed publicly. One approach to protecting such data is to require that the program’s be certified using a security type system. Security type systems have, previously, only been studied for programs written entirely in a single language. In practice, however, software is frequently implemented using multiple specialized languages for different tasks.

We present a framework that facilitates reasoning over composed languages. In it guarantees of sufficiently related component languages can be lifted to the composed language. This can significantly lower the burden necessary to certify that such composite programs are safe. Our simple but powerful approach relies, at its core, on computability and security-level separability. To demonstrate this framework, we develop a secure type system for an expressive fragment of SQL and embed it into a secure While language from the literature. We then prove all the requirements necessary to apply our approach and show that all well-typed composite While+SQL programs are safe.

1 Introduction

In contemporary software engineering practice, developers often use several different languages while implementing large systems. This ranges from using separate languages in different files (e.g., configuration files in XML or libraries coded in different languages) to using code in one language that is directly embedded in the code of another language. As an example, one commonly embedded language is SQL, which is used to declaratively retrieve data. Such languages are common even in the mobile app space: both Android and iOS expose bindings for SQL systems (e.g., SQLite) to applications written primarily in Java and Objective C, respectively.

At the same time, privacy is becoming an increasingly important property of software. Mobile platforms like smartphones and tablets, for example, are built around “app stores”, where users can download new applications. These applications can gain access to private information that is stored on the device, like contacts, phone logs, location information, etc. In the mobile space, major vendors have chosen two ways to help users: either, a simple form of static permissions, as used on the Android platform, or a review of every application,

as in the case of iOS. Neither approach is very satisfying: several cases of apps that violate the privacy of the owner’s device have become known recently for both platforms. A formal and sound mechanism that can be applied to formally verify such software against privacy policies would greatly improve this situation. Type systems enforcing noninterference can provide such a mechanism.

Noninterference [5] ensures that any compliant program cannot leak private information to public channels. Type systems are a common method to guarantee noninterference, e.g., [19, 6, 13, 20, 1, 7], for an overview see [18], and the approach has been extended to cover entire distributed systems [10]. In all of this work, however, exactly one language is treated.

In contrast, how can we (statically) guarantee the safety of programs that are composed from elements in different languages? We propose to compose security-typed languages into *composed languages*, such that well-typed programs in a composed language can be guaranteed to comply with noninterference. This paper studies an approach that, under certain assumptions, makes it possible to leverage proofs of non-interference of well-typed host language and well-typed embedded language programs to prove noninterference of well-typed composed language programs. In order to generalize this composition over security-typed host and security-typed embedded languages that use different proofs that well-typed programs are noninterferent, our approach relies on host languages being complete with respect to being able to compute any noninterferent function over its data types. This allows us to establish that executing noninterferent code does not introduce any behaviors that could not be observed in the host language.

This paper validates the approach on a composition of a security-typed While language (derived from [19]) and a simple security-typed SQL fragment (of the data-manipulation sublanguage). We demonstrate a constructive technique to prove completeness with respect to noninterfering computations on the example of the While language, and formally prove all requirements to complete our framework approach and formally prove While+SQL secure.

The contributions of this paper include

- describing a general framework for composition of security-typed languages, such that noninterference of the composed language can be established from the proofs of noninterference of the component languages
- demonstrating the framework on a composition of a security-typed While language and a security-typed SQL fragment
- formalization and soundness for a security-typed SQL fragment including joins and nested queries

This paper is structured as follows. Section 2 gives a short overview over background material. Next, in Section 3 we detail our goals of composition on the example of a student information system. Our framework approach is outlined in Section 4 and compared against traditional techniques used to establish soundness of security type systems. Section 5 formalizes the While and SQL languages we compose in this paper, and describes the extensions that create the composed language. The following section then applies our framework to this composition. Section 7 compares against some related work, while Section 8 concludes.

2 Background

Most security type systems are based on the lattice model for security [3], that is, security information is taken from a lattice of security “levels”. This makes it possible to easily formulate flows of information, i.e., information is allowed to flow from a lower level to a higher level; and also helps to define the abilities of an attacker, i.e., an attacker is allowed to observe all information or actions below a certain level. Informally, (1) computations that involve private=high information need to be classified private, (2) we cannot allow direct assignments of private information to public storage, and (3) we cannot allow indirect leaks, that is, assignments to public storage in a private context.

The most simple non-trivial lattice used is $\{L, H\}$ with the obvious ordering ($L \sqsubseteq L$, $L \sqsubseteq H$, $H \sqsubseteq H$), but many more complicated lattices have been proposed (e.g., [14]). We only focus on lattice-based security in our framework. Noninterference can be generalized with the help of an indistinguishability relation that defines which parts of values may be distinguishable (and thus should not be influenced).

3 Motivation

Our ultimate goal is to *prove* safe the composition of practical languages. We use the example of a system composed of application logic and backend storage here. The application logic is written in an imperative language, while the storage is accessed with a SQL dialect. For example, imagine a university system that stores students’ data. We can model this with a table that stores a record for each student, e.g., the student’s name, room number, and several grades. Mandated by law, the grades are private information and must not be shared with unauthorized personnel, while room numbers can be used in a university directory. We can model this security by assigning low confidentiality to the name and room number, and high confidentiality to the grades. Now general staff can be classified as low, too, so to be able to access a student’s room number. We can write a program that reads the database and writes this information to a low output, e.g., a generally accessible website. Note that `eval` will process the nested query and return the result. In a more practical language, the explicit use of this construct may be hidden by a layer of syntactic sugar.

```
Program list-students;
Schema:  students: name=L, room=L, grade1=H;
Code:
  length{L} = eval("SELECT count(*) FROM students");
  i{L} = 0;
  while i < length do
    name{L},room{L},grade1{L} = eval("SELECT name,room, grade1
      FROM students LIMIT $x,$x",i);
    print-public name, room, grade1
    i++;
```



Fig. 1. Flow in Composed Program & Simulated Program

This program should be rejected because of the leak of the grade, namely that the level in the host language(L) does not correspond with the level in the embedded language(H). However, if that part is removed, the program is valid and should pass the information flow checker. Similarly, updates in the database need to be protected, e.g., the following program needs to be rejected.

```

Program update-room;
Schema:  students: name=L, room=L, grade1=H;
Code:
grade1{H} = eval("SELECT grade1 FROM students WHERE name='...'");
if grade1>3 then
  eval("UPDATE room=3 WHERE name='...'");

```

Assume that our SQL type system is proven secure (see Section 5.2), as well as the While language (e.g., [19]). We would now like to prove that the composed language is also secure.

4 Framework for Composition

Our framework is focused on computability and centered around the three following issues: (1) Simulation, (2) Typability, and (3) Replacement. Graphically, we would like to transform a program as seen in Figure 1(a) to the one in Figure 1(b). If the latter program is equivalent to the former and typable, then the safety result of the host language will guarantee noninterference, which then also holds for the former.

4.1 Simulation

Our general approach is to establish that executing noninterfering embedded code does not introduce any behaviors that could not be observed in the host language. In that case, the host language is expressive enough for all embedded-language behaviors, which will be treated by the type system and should be decided to be secure. For technical reasons we split this property into two sufficient components. The first is simulation.

Let us assume that the host language is computationally at least as powerful as the embedded language. For example, let the host language be Turing-complete, that is, every Turing-computable function can be computed by a host-language program, and the embedded language might be Turing-computable, that is, every embedded-language program computes a Turing-computable function. Then the host language can obviously simulate any embedded-language

program. Note that any other specific level of computability is acceptable, as long as it allows covering all embedded-language behaviors.

While the simulation property allows us to lift *all* computations of the embedded language, we are only interested in noninterfering ones. If a composed program is typed, then the definition of `eval` and its typing rules ensure that the embedded code fragment is typable in the embedded language. This means that the embedded program is noninterfering by virtue of the embedded language’s noninterference statement. Now since the simulation is required to compute an equivalent function, and noninterference is a semantic property that only relies on inputs and outputs, the simulation is also noninterfering. Note that this of course only holds if the interface established by `eval` satisfies constraints about the values and types involved, for example, preserving indistinguishability of values.

Finally, note that only this step is specific to a certain composition. For each composition, it needs to be ensured that the embedded language is at most as powerful as the host language. All following steps are specific to the host language and can be reused for other compositions. However, we think that the simulation step is broadly applicable, because most host languages are expected to be Turing-complete general-purpose languages that are at the top of a realistic computability hierarchy.

4.2 Typability

While we now know that the simulation itself is noninterfering, it remains to see whether this holds for the whole program the simulation is a part of. We reduce this problem to two steps, the first of which is typability. If we can show that some simulation is typable, host-level noninterference will apply and secure the computation.

In general, the framework is a theoretical tool to prove composed languages secure. As such, the actual simulation is not important for running the composed program. This means that we can use *any* simulation that is equivalent to the embedded code fragment, no matter how complex, if it helps us find a typing.

We define the following property.

Definition 1 (Security Completeness). *A security-typed language \mathcal{L} is security-complete if and only if for every (not necessarily typable) program c that computes a function f , where f is noninterfering with respect to security signature \mathcal{S} , there exists a program c' that also computes f and is typable corresponding to \mathcal{S} .*

In the case of a security-complete host language, the typability step is immediately obvious, since the simulation is noninterferent. We will then use the typable simulation program guaranteed to exist. The While language we investigate in this paper is security-complete, as we will show in Theorem 2.

Note that all practical general-purpose security-typed languages seem to be security-complete for functions over integers. In other work [4] we study the limits of security-completeness and basic requirements on a security type system. We derive sufficient conditions for versions of security-completeness for functional

languages with nonrecursive and recursive algebraic datatypes, as well as languages with heaps and side effects including a simple class-based object-oriented language.

4.3 Replacement

Last, if we can replace all instances of `eval` in the composite program with the corresponding simulations, and can show that typings and meaning are properly preserved, we have found a pure host-language program that is host-typable, which implies its noninterference. Again, noninterference can then be reflected onto the composite program, which is functionally equivalent. This is the third step, which is also only host-language specific.

Note that this is not a traditional substitution lemma. Standard substitution lemmas are concerned with replacing a variable with a term, and preserving a typing in the process. In our case, a whole construct, namely `eval`, must be replaced, and the typing of the replaced term may be under a different environment to account for the possible temporaries and side effects of the simulation. Furthermore, one has to show that the programs before and after substitution are functionally equivalent. However, as mentioned, this has to be proven only once and can be reused for other compositions involving this host language.

4.4 Applicability

One might ask if embedding a less powerful language is useful at all, and thus if our approach is realistic in practice. For one, in practice many embedded languages are only powerful in certain domains (i.e., domain specific languages) and excel in conciseness and expressivity there, while general-purpose languages are usually Turing-complete and can do the same work.

4.5 Limitations of Proof Manipulation

Last, we would like to compare our approach against a more traditional one. A strawman approach to proving noninterference for well-typed composed programs is by defining a mechanical procedure for directly manipulating the proofs of noninterference of programs in the host and embedded languages. However, this seems to be tied closely to the format and details of the noninterference proof of the host language. For example, if that proof was done syntactically via a subject reduction theorem, as for example in [15], that theorem would need to be extended. Subject reduction is usually shown by some inductive argument, for example over the input typing. As such, we could extend the case analysis of the induction with a specific argument for `eval` that is derived from the typing constraints and hope to get a well-formed proof for the composed language.

However, the picture is not that simple. While adding a case to subject reduction is simple and (mostly) independent from the other constructs, other cases may use their own nested inductive arguments, auxiliary lemmas, or inversions. [15], for example, uses auxiliary lemmas for weakening, projection, and

substitution, with all lemmas ranging over *all* constructs in the language. Without a detailed knowledge of the proof and the statements necessary, it seems impossible to generically prove correctness by manipulating an unknown proof.

5 Languages

This section formalizes the languages we will use for composition. In Subsection 5.1 we introduce While. The following subsection formalizes our fragment of SQL, gives its type system, and proves it sound (Theorem 1). Finally, Subsection 5.3 formalizes the composed language.

5.1 Host: WHILE

We base our exposition on a simplified version of the work of Volpano et.al [19]. This is a simple While language, with the following expressions and statements.

$$e ::= n \mid x \mid e \odot e \quad c ::= x := e \mid c; c' \mid \text{if } e \text{ then } c \text{ else } c' \mid \text{while } e \text{ do } c$$

The language only supports integer values. A state μ binds variables to integers. We use a big-step semantics. Expressions are side-effect free, and evaluated by $\mu(e)$, which is defined in the obvious way. The semantics connects an input state and a statement with an output state and is fully standard and thus elided here.

Since the language only supports integers, it is not necessary to have a ground type system. A typing environment Γ binds variables to security levels. Judgments have the form $\Gamma \vdash e : \ell$ and $\Gamma \vdash c : \ell \text{ ok}$ and are also standard.

Two states are indistinguishable with respect to a typing if they agree on all observable variables: $\mu_1 \sim_{\Gamma, \ell} \mu_2 \iff \forall x. \Gamma(x) \sqsubseteq \ell \implies \mu_1(x) = \mu_2(x)$. Noninterference states that a typed computation, when started with indistinguishable states, results in indistinguishable states. For a proof we refer to [19].

$$\begin{aligned} \forall \ell, \Gamma, c, \mu_1, \mu_2, \mu'_1, \mu'_2. \Gamma \vdash c : \ell \text{ ok} \wedge \mu_1 \sim_{\Gamma, \ell} \mu_2 \wedge \mu_1, c \Downarrow \mu'_1 \wedge \mu_2, c \Downarrow \mu'_2 \\ \implies \mu'_1 \sim_{\Gamma, \ell} \mu'_2 \end{aligned}$$

5.2 Embedded: SQL

We formalize a simplified version of the data retrieval and manipulation fragment of SQL. We allow (finitely many) named tables of integer data. Names \hat{t} are drawn from a set \mathcal{T} . The set of tables is static and finite for a particular program: we do not allow table creation or deletion. Tables have a finite number of columns $\mathcal{I}_{\hat{t}}$, where we assume for simplicity that column names are distinct between tables. We use i to range over *all* column names. We have the following syntax, where we use a dot to distinguish from the WHILE syntax.

$$\begin{aligned} \hat{t} &::= \hat{t} \mid \hat{t} \text{ join } \hat{t}' \text{ on } i = i' & \dot{e} &::= \dot{n} \mid i \mid \dot{x} \mid \dot{e} \oplus \dot{e}' \\ \dot{c} &::= \text{select } \dot{e}_1, \dots, \dot{e}_n \text{ from } \hat{t} \text{ where } \dot{e}' \mid \text{insert } i = \dot{e} \text{ into } \hat{t} \mid \\ &\quad \text{update } i = \dot{e} \text{ in } \hat{t} \text{ where } \dot{e}' \mid \text{delete from } \hat{t} \text{ where } \dot{e} \end{aligned}$$

A table state T is represented as a finite map of naturals to records r , which are a total map of column names to integers. Note that we identify states up to an order-preserving remapping. A database state ν is a finite map of table names to table states. For simplicity of handling table joins we extend the set of table names by table expressions, e.g., if A and B are table names and i and i' column names, then $A \text{ join } B \text{ on } i = i'$ is a table name.

Expressions can be evaluated for a record, denoted by $r(\dot{e})$, in the obvious way. Expressions do not change the database. Table expressions project and manipulate table states from a database state in the obvious way.

A natural semantics reduces a statement and database state to a result and a state. We use the following auxiliary definitions.

$$T_{\dot{e}}^{\neq} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) \neq \dot{0} \\ \perp & \text{otherwise} \end{cases} \quad T_{\dot{e}}^{\equiv} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) = \dot{0} \\ \perp & \text{otherwise} \end{cases}$$

$$(i_1 : \dot{n}_1, \dots).i_j \leftarrow \dot{n}' = (i_1 : \dot{n}_1, \dots, i_j : \dot{n}', \dots)$$

$$T_{i, \dot{e}, \dot{e}'}^{\leftarrow} = \lambda n. \begin{cases} r.i \leftarrow r(\dot{e}) & T_{\dot{e}'}^{\neq}(n) = r \\ T(n) & \text{otherwise} \end{cases}$$

Here $T_{\dot{e}}^{\neq}$ restricts the domain of T to the records satisfying \dot{e} , while $T_{\dot{e}}^{\equiv}$ restricts to the opposite. Finally $T_{i, \dot{e}, \dot{e}'}^{\leftarrow}$ computes a new map where columns i of records r that satisfy \dot{e}' are updated to $r(\dot{e})$. Now with the intent that $\perp(\dot{e}) = \perp$ the semantic rules are defined as follows.

$$\nu, \text{select } \dot{e} \text{ from } \dot{t} \text{ where } \dot{e}' \Downarrow \nu, \lambda n. \left(\nu(\dot{t})_{\dot{e}'}^{\neq}(n)(\dot{e}_1), \dots \right)$$

$$\begin{aligned} &\nu, \text{insert } i_j = \dot{n} \text{ into } \dot{t} \Downarrow \\ &\quad \nu[\dot{t} := \nu(\dot{t}) + (i_1 : \dot{0}, \dots, i_j : \dot{n}, \dots)], \emptyset \\ &(\text{where } + \text{ extends } \nu(\dot{t})'s \text{ domain}) \end{aligned}$$

$$\nu, \text{update } i = \dot{e} \text{ in } \dot{t} \text{ where } \dot{e}' \Downarrow \nu[\dot{t} := \nu(\dot{t})_{i, \dot{e}, \dot{e}'}^{\leftarrow}], \emptyset$$

$$\nu, \text{delete from } \dot{t} \text{ where } \dot{e} \Downarrow \nu[\dot{t} := \nu(\dot{t})_{\dot{e}}^{\equiv}], \emptyset$$

We assign security levels to tables for two reasons. First, all tables have a simple level that describes the confidentiality of the table itself. Observers not at or above this level cannot access a table at all. Second, we assign security levels to columns, that is, all records have the same security level for the corresponding columns. We denote the mapping of column names of a table to security levels by δ , and $\ell_\delta = \prod \delta(i)$. For simplicity, we map the table security level to the synthetic column name *table*.

For typing purposes, we collect all δ in Δ which maps from table names. For table expression names, we define the following evaluation.

$$\Delta(\dot{t}_1 \text{ join } \dot{t}_2 \text{ on } i_1 = i_2) = \lambda i. \begin{cases} \bigsqcup_{i \in \{1,2\}} (\Delta(\dot{t}_i)(\text{table}) \sqcup \Delta(\dot{t}_i)(i_i)) & i = \text{table} \\ \Delta(\dot{t}_1)(i) & i \in \Delta(\dot{t}_1) \\ \Delta(\dot{t}_2)(i) & i \in \Delta(\dot{t}_2) \wedge i \notin \Delta(\dot{t}_1) \\ \perp & \text{else} \end{cases}$$

The type system is defined by the following sets of rules, where $\dot{\Gamma}$ is a typing environment mapping variables to security levels. Note that typing of expressions refers to a table typing δ , whereas statement typing uses Δ . Well-formedness requirements for δ and Δ are standard.

$$\begin{array}{c} \delta, \dot{\Gamma} \vdash \dot{n} : \ell \quad \delta, \dot{\Gamma} \vdash i : \delta(i) \quad \delta, \dot{\Gamma} \vdash \dot{x} : \dot{\Gamma}(\dot{x}) \\[10pt] \frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2}{\delta, \dot{\Gamma} \vdash \dot{e} \oplus \dot{e}' : \ell_1 \sqcup \ell_2} \quad \frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \ell'}{\delta, \dot{\Gamma} \vdash \dot{e} : \ell'} \\[10pt] \frac{\delta = \Delta(\hat{t}) \quad \forall i. \delta, \dot{\Gamma} \vdash \dot{e}_i : \ell_i \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell \quad \ell \sqcup \delta(\text{table}) \sqsubseteq \ell'}{\Delta, \dot{\Gamma} \vdash \text{select } \dot{e}_1, \dots, \dot{e}_n \text{ from } \hat{t} \text{ where } \dot{e}' : (\ell_1, \dots, \ell_n)^{\ell'} / \top \text{ ok}} \\[10pt] \frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \text{insert } i = \dot{e} \text{ into } \dot{t} : \perp^\perp / \ell_\delta \text{ ok}} \quad \frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \delta(\text{table})}{\Delta, \dot{\Gamma} \vdash \text{delete from } \dot{t} \text{ where } \dot{e} : \perp^\perp / \ell_\delta \text{ ok}} \\[10pt] \frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \text{update } i = \dot{e} \text{ in } \dot{t} \text{ where } \dot{e}' : \perp^\perp / \delta(i) \text{ ok}} \end{array}$$

For SQL we have to formalize to notions of indistinguishability. We use a projection to erase all information in a record that is not typed at or below ℓ , and lift it to sequences.

$$\downarrow_\ell^\delta(r).i = \begin{cases} r.i & \delta(i) \sqsubseteq \ell \\ 0 & \text{otherwise} \end{cases} \quad \downarrow_\ell^\delta(f) = \lambda n. \downarrow_\ell^\delta(f(n))$$

We define two result sets as indistinguishable with respect to a type $\ell_\bullet^{\ell'}$ (where ℓ_\bullet defines a sequence of labels), abusing notation of projection, as $s_1 \sim_{\ell_\bullet^{\ell'}, \ell_o} s_2 \iff \ell' \sqsubseteq \ell_o \implies (|s_1| = |s_2| \wedge \downarrow_{\ell_o}^{\ell_\bullet}(s_1) = \downarrow_{\ell_o}^{\ell_\bullet}(s_2))$. Projection is also used to define indistinguishability of two table states, which are indistinguishable with respect to observer level ℓ if they agree on all columns at most ℓ : $T_1 \sim_{\delta, \ell} T_2 \iff \delta(\text{table}) \sqsubseteq \ell \implies \downarrow_\ell^\delta(T_1) = \downarrow_\ell^\delta(T_2)$. Finally, indistinguishability of table states is lifted component-wise to database states: $\nu_1 \sim_{\Delta, \ell} \nu_2 \iff \forall t \in \mathcal{T}. \nu_1(t) \sim_{\Delta(t), \ell} \nu_2(t)$ We can now state noninterference for SQL.

Theorem 1 (SQL Noninterference).

$$\begin{aligned}
& \forall \Delta, \ell_\bullet, \ell', \ell'', \ell_x, c, \mu_1, \mu_2, \mu'_1, \mu'_2, \dot{n}_1, \dot{n}_2, s_1, s_2. \\
& \Delta, x : \ell_x \vdash c : \ell_\bullet^{\ell'} / \ell'' \text{ ok} \wedge \mu_1 \dot{\sim}_{\Delta, \ell_o} \mu_2 \wedge \dot{n}_1 \dot{\sim}_{\ell_x, \ell_o} \dot{n}_2 \wedge \mu_1, c[x := \dot{n}_1] \Downarrow \mu'_1, s_1 \wedge \\
& \mu_2, c[x := \dot{n}_2] \Downarrow \mu'_2, s_2 \\
& \implies \mu'_1 \dot{\sim}_{\Delta, \ell_o} \mu'_2 \wedge s_1 \dot{\sim}_{\ell_\bullet^{\ell'}, \ell_o} s_2
\end{aligned}$$

Remarks The SQL fragment here is simplified, but one can regain many SQL features when compositing. For example, inserting a full record can be implemented by first inserting a dummy record and then updating its columns. Other features, like special functions and selection modifiers, can be easily added.

The type system might seem very restrictive, but this is required for soundness. An attacker may run the query `select 1 from t where 1`, which yields a result set the size of the number of rows, potentially leaking information about the insertions and deletions. Note that confidential deletion and insertion can still be modeled: a confidential column can signal whether a record is valid.

5.3 Composed Language

This section formalizes the extension of the host language, to create a composed language of WHILE and SQL. We restrict ourselves to transfer simple integers. We add the statement $x_1, \dots, x_n := \text{eval } x' \text{ in } \dot{c}$ for evaluation, where x_i designate the variables that will hold the result, and x' is a host-level variable that is a parameter for the embedded computation \dot{c} . We will use x_\bullet to denote a list of variables.

We extend the host semantics in the following two ways. First, the embedded state ν is threaded through the original reduction rules, which do not update the embedded states themselves. In the case of WHILE, this is unambiguous. In languages with multiple nested reductions this threading will induce a certain evaluation order. Second, reduction of `eval` is given by the following new rules.

$$\begin{aligned}
& \frac{\nu, \dot{c}[x := \mu(x')] \Downarrow \nu', \emptyset}{\mu, \nu, x_\bullet := \text{eval } x' \text{ in } \dot{c} \Downarrow \mu[x_1 := 0][\dots][x_n := 0], \nu'} \\
& \frac{\nu, \dot{c}[x := \mu(x')] \Downarrow \nu', s \quad s \neq \emptyset \quad s(0) = (n_1, \dots, n_n)}{\mu, \nu, x_\bullet := \text{eval } x' \text{ in } \dot{c} \Downarrow \mu[x_1 := n_1][\dots][x_n := n_n], \nu'}
\end{aligned}$$

Typing rules are changed accordingly. That is, Δ is threaded through the original typing rules, and statement types are extended by the lower bound of changes in the embedded state. Furthermore, we add the following typing rule for `eval`.

$$\frac{\Gamma \vdash x' : \ell \quad \Delta, [x : \ell] \vdash \dot{c} : \ell_\bullet^{\ell_2} / \ell_s \text{ ok} \quad \forall i. \ell_i \sqcup \ell_2 \sqsubseteq \Gamma(x_i)}{\Gamma, \Delta \vdash x_\bullet := \text{eval } x' \text{ in } \dot{c} : (\prod \Gamma(x_i)) \sqcap \ell_s \text{ ok}}$$

Indistinguishability is lifted component-wise, that is, $\mu_1, \nu_1 \dot{\sim}_{\Gamma, \Delta, \ell} \mu_2, \nu_2 \iff \mu_1 \sim_{\Gamma, \ell} \mu_2 \wedge \nu_1 \dot{\sim}_{\Delta, \ell} \nu_2$. This suffices because SQL values, that is, result sets, do

not occur as values in the composed language. Then noninterference is stated analogously to the WHILE case.

$$\begin{aligned} & \forall \ell, \Gamma, c, \mu_1, \nu_1, \mu_2, \nu_2, \mu'_1, \nu'_1, \mu'_2, \nu'_2. \\ & \Gamma \vdash c : \ell \text{ ok} \wedge \mu_1, \nu_1 \dot{\sim}_{\Gamma, \Delta, \ell} \mu_2, \nu_2 \wedge \mu_1, \nu_1, c \Downarrow \mu'_1, \nu'_1 \wedge \mu_2, \nu_2, c \Downarrow \mu'_2, \nu'_2 \\ & \implies \mu'_1, \nu'_1 \dot{\sim}_{\Gamma, \Delta, \ell} \mu'_2, \nu'_2 \end{aligned}$$

5.4 Remarks

In general, our framework can support more complicated compositions. We allow auxiliary functions α , α^τ , γ and γ^τ , establishing how certain types and their values can be translated. We require monotonicity with respect to security levels and non-decreasing when composed for the type translations α^τ and γ^τ , and preservation of indistinguishability for value translations α and γ . For our example, both α and α^τ are simply identity functions. γ takes a list of tuples and projects it such that the result is the first tuple, if such exists, or zeros otherwise. γ^τ takes the element and length labels of SQL and combines them into their upper bound to account for the type of the data values, as well as for the result being empty or not. \sqcup is monotonous, so γ^τ is. Indistinguishability of two lists of tuples implies that they are either both empty or agree on their first element (or observable components thereof). Thus, indistinguishable results sets will be projected to indistinguishable tuples at the host level.

6 Framework Proofs

This section applies our framework approach outlined in section 4 and formally verifies that the composition of While and SQL from the previous section is safe, that is, typable composed programs are noninterfering. For this, recall the three main steps:

1. Embedded-language programs can be simulated in the host
2. The simulation is noninterferent and can be typed
3. Replace embeddings with the simulation; the now pure-host program is typable, implying noninterference of the composed program

6.1 Simulation

For the first step, we need to establish that the host language is computationally at least as powerful as the embedded language. In our case, we note that WHILE is Turing-complete, and SQL is Turing-computable. Thus, any program c can be simulated by a program c .

6.2 Preservation of Noninterference

The second step of our approach is split into two parts: noninterference and typability. Noninterference is derived from the conditions of the eval. Namely, given indistinguishable inputs, the respective values of the parameter will be indistinguishable. This is preserved when switching to the embedded language. By noninterference theorem of the embedded language and the embedded fragment being well-typed, the outputs must be indistinguishable. The translation to the host level preserves this property. The final update of the host state will thus maintain state indistinguishability. Overall, we have that indistinguishable inputs to an eval lead to indistinguishable output.

Noninterference is a semantical property defined over inputs and outputs. Since the simulation is functionally equivalent, that is, produces the same outputs for the same inputs, the simulation is also noninterferent.

6.3 Typability

We now need to show that the simulation is typable with respect to the types of the eval. That is, we need to type the simulation such that the input corresponding to the eval's parameter is typed the same, i.e., as $\Gamma(x)$, as well as all the column encodings according to the typing given by Δ . Note that, in fact, it is not even necessary to show typability of the simulation found in the first step. In many languages, many programs compute the same function. Only one program out of the class of equivalent programs needs to be typable. We use the following theorem to posit the existence of such a program.

Theorem 2 (Security Completeness). *If a function f is computable in WHILE, and noninterferent under a signature given by Γ , then there exists a WHILE program c that is typable under a typing environment Γ' that is an extension of Γ .*

We only sketch the proof here, which relies on two auxiliary lemmas. First, any WHILE program is typable at a single level ℓ . Second, two typed WHILE programs can be composed such that the composition is typable. Equipped with these lemmas, we construct a typable program for every output of the original program at that output's level, which is possible by the first lemma. We can substitute the inputs at a higher security level with arbitrarily chosen constants, e.g., zero. Noninterference guarantees that the output remains correct. Finally, we can compose all separate programs to compose the whole output, as validated by the second lemma.

6.4 Replacing eval

The last step of our approach ties the previous subsections together and shows how to replace the eval for the simulation.

Lemma 1 (Substitution Typable). *Assume a context $E[\bullet]$, that is, a composed program with a statement hole. Furthermore assume an $x := \text{eval } x' \text{ in } \dot{c}$, Γ , Δ and ℓ such that $\Gamma, \Delta \vdash E[\text{eval } x' \text{ in } \dot{c}] : \ell \text{ ok}$. Then there exists an extension Γ' of Γ such that $\Gamma', \Delta \vdash E[c_{\text{eval}}] : \ell \text{ ok}$.*

We can use this lemma to iteratively replace all eval statements in the original composed program with their respective simulations. The result is a typing of a pure-host statement under the composed-language rules. The next lemma states that such a typing induces the corresponding host-level typing of the statement.

Lemma 2 (Eval-free Composed To Host). *If a statement c that does not contain eval is typed under Γ and Δ as $\Gamma, \Delta \vdash c : \ell \text{ ok}$, then c can be typed as $\Gamma \vdash c : \ell \text{ ok}$.*

Corollary 1 (Simulation Pure-Host Typable). *If c is a composite program that is typable as $\Gamma, \Delta \vdash c : \ell \text{ ok}$, then there exists an extension Γ' of Γ that encodes Δ such that the simulation program c_{eval} , where all eval statements have been substituted for their simulation, is typable as $\Gamma' \vdash c_{eval} : \ell \text{ ok}$.*

This formally proves that the simulation program is noninterferent by noninterference of typed host-language programs. Now, the final step needs to formally show that the simulation program is equivalent to the original program. This is obviously modulo the behavior of temporaries of the simulation, which are exposed to the host.

Theorem 3 (Simulation Equivalence up to Γ). *If $\Gamma, \Delta \vdash c : \ell \text{ ok}$, then there exists an extension Γ' such that $\Gamma' \vdash c_{eval} : \ell \text{ ok}$, and for all $\mu_1, \mu'_1, \nu, \nu', \mu_2$ where $\Gamma \vdash \mu_1 \text{ ok}$, $\Gamma' \vdash \mu_2 \text{ ok}$ and μ_2 is an extension of μ_1 such that the extension encodes ν , and $\mu_1, \nu, c \Downarrow \mu'_1, \nu'$, then there exists μ'_2 an extension of μ'_1 such that $\mu_2, c_{eval} \Downarrow \mu'_2$ and the extension encodes ν' .*

Corollary 2 (Replacement). *For a typable composed program c , the program c_{eval} created by replacing all eval statements with a corresponding simulation, is typable and functionally equivalent to c .*

This concludes all three steps of our framework. Together, this formally shows that all composed-language programs can be translated into pure host-language programs that remain typed with a corresponding typing environment and are functionally equivalent to the composed program. Noninterference of the host language applies to typed host programs, so the translation is noninterfering. Since the composed program computes the same function, it is also guaranteed to be noninterfering.

7 Related Work

To the best of our knowledge, this is the first work to make use of completeness in the context of security-typed languages, which we studied in [4]. Kahrs [8] studied completeness for basic type systems, where the question is if all computable functions that are “well-going” can be typed. Kahrs uses transitions systems, whereas our goal is to permit easy adoption of existing languages.

Traditional work in security-typed languages attempts to broaden the permissibility of the type system, that is, accept more programs as typed and thus

secure. For an overview we refer to [18]. Our work is orthogonal to such efforts. We show that, under certain constraints, there are always programs that compute a given noninterferent function.

Basic type-safety of composition has been investigated, e.g., Bierman et.al. [2] studied an SQL-like extension to C#. However, the query language was fully incorporated into the host, which seems infeasible for the sheer amount and expressivity of languages considered for embedding in practice. Compositionality of noninterference has been studied in different contexts (e.g., [12, 17, 11] for overviews). The goal here is to derive constraints on when composing secure program fragments of *one* formal system yields a secure result. In contrast, our fragments are derived from *different* security-typed languages (here While and SQL), and we base our compositionality result around the notion of *security completeness*.

There are several related approaches to secure a complex system. Li and Zdancewic [9] describe a web programming language extended with an interface to a relational database (note only simple select statements), where the language and interface are security typed. But this means that the storage side needs to be fully trusted. In a similar vein, arrows and monads (e.g., [16] and references therein) can be used to isolate and control information flows in a library fashion. We note that a library approach to embedded languages restricts expressivity and conciseness to that of the host language.

Fabric [10] extends JIF to a secure distributed system, where storage is an integral part. We believe, however, that separation of concerns is important and our approach allows a modular proof of the safety of the whole system, composed from smaller fragments. Also note that, persistent storage is but one use of an embedded language, and it seems unrealistic to expect one language to excel in all domains.

8 Conclusion & Future Work

In this paper, we have presented a framework that can be used to show the security of a language that is composed from host and embedded languages that are themselves secure. The approach is based on a simulation of the embedded component's behavior and the accompanying proof that the noninterference of the component's computation guarantees that there exists a typable program for the simulation. We can then refer to the strong guarantees of the host language to prove a composite program noninterferent.

This result allows us to develop separate type systems for individual languages, and lift the results to compositions of languages. It thus significantly reduces the burden of showing security in modern programs that employ many programming languages for different tasks like data retrieval and modification. In this paper we have shown safe the composition of a While language and the data manipulation fragment of SQL. Our effort for the While language can be reused, because our framework can be instantiated for other embedded languages with minimal effort.

References

1. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. *J. Funct. Program.* 15(2), 131–177 (Mar 2005)
2. Bierman, G., Meijer, E., Schulte, W.: The essence of data access in *cw*: The power is in the dot. In: In ECOOP’02 (2002)
3. Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* 19, 236–243 (May 1976)
4. Gampe, A., Von Ronne, J.: Security completeness: Towards noninterference in composed languages. In: PLAS ’13 (2013), conditionally accepted
5. Goguen, J.A., Meseguer, J.: *Security Policies and Security Models*, vol. pages, pp. 11–20. IEEE (1982)
6. Heintze, N., Riecke, J.G.: The slam calculus: programming with secrecy and integrity. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL ’98 (1998)
7. Hunt, S., Sands, D.: From exponential to polynomial-time security typing via principal types. In: Proceedings of the 20th European conference on Programming languages and systems. ESOP’11/ETAPS’11 (2011)
8. Kahrs, S.: Well-going programs can be typed. In: Proceedings of the 6th international conference on Typed lambda calculi and applications. TLCA’03 (2003)
9. Li, P., Zdancewic, S.: Practical information-flow control in web-based information systems. In: Proceedings of the 18th IEEE workshop on Computer Security Foundations. CSFW ’05 (2005)
10. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: a platform for secure distributed computation and storage. In: SOSP ’09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (2009)
11. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium. CSF ’11 (2011)
12. McCullough, D.: Specifications for multi-level security and a hook-up. *Security and Privacy, IEEE Symposium on* 0, 161 (1987)
13. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL) (1999)
14. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proc. 17th ACM Symp. on Operating System Principles (1997)
15. Pottier, F., Simonet, V.: Information flow inference for ML. In: POPL ’02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 319–330. ACM, New York, NY, USA (2002)
16. Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in haskell. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 13–24. Haskell ’08, ACM, New York, NY, USA (2008)
17. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. *J. Comput. Secur.* 9(1-2), 75–103 (Jan 2001)
18. Sabelfeld, A., Myers, A.: Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on* 21(1), 5 – 19 (Jan 2003)
19. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. *J. Comput. Secur.* 4, 167–187 (January 1996)
20. Zdancewic, S., Myers, A.C.: Secure information flow and cps. In: ESOP ’01: Proceedings of the 10th European Symposium on Programming Languages and Systems. pp. 46–61. Springer-Verlag, London, UK (2001)

A Formal Framework for Secure Routing Protocols

Chen Chen¹ Limin Jia² Hao Xu¹ Cheng Luo¹
Wenchao Zhou³ Boon Thau Loo¹

¹ University of Pennsylvania, Philadelphia, PA 19104, USA
chenche, haoxu, boonloo@cis.upenn.edu

² Carnegie Mellon University, Pittsburgh, PA 15213, USA liminjia@cmu.edu

³ Georgetown University, Washington, DC 20057, USA wzhou@cs.georgetown.edu

Border Gateway Protocol (BGP), the de facto Internet routing protocol, is vulnerable to various attacks. Redesigns of Internet routing infrastructure (e.g. S-BGP and SCION) have been proposed to address the security concerns with BGP. However, none of them formally verifies its security claims. Existing model-checking-based protocol analysis tools cannot be directly applied to verifying routing protocols, as it requires verification on an infinite number of network topologies. Proving small model theorems enables sound and complete verification on a finite number of topologies [1]; however, such theorems are specific to each protocol and may not exist for some protocols.

We develop a unified formal framework that combines development, empirical evaluation, and formal verification of secure routing protocols. The framework uses SeNDLog, a declarative networking language resembling distributed Datalog, as the protocol specification language. The specification can both be translated to executable code for performance evaluation and be used to generate proof obligations for verification.

We develop trace-based operational semantics for SeNDLog. The semantics adopts a distributed execution model. Network states are modeled as relational databases maintained at each node. Evaluation of SeNDLog programs dictates how nodes communicate with each other and how tuples in databases are updated incrementally.

Inspired by prior work on analyzing safety properties of programs executing concurrently with adversaries [2], we develop a sound Hoare-style program logic for SeNDLog. After specifying security properties in first-order logic, we use our program logic to derive invariant properties of the SeNDLog program by checking that each rule in that program maintains those invariant properties.

We implement a compiler for SeNDLog, and a verification condition generator (VCG) that extracts lemmas necessary to verify a given SeNDLog program. Verifying a secure routing protocol within our framework involves (1) encoding the protocol in SeNDLog; (2) specifying the security properties of the protocol and auxiliary properties of the program; (4) using VCG to generate proof obligations in a theorem prover (e.g. Coq); and (5) discharging the proof obligations. We encode S-BGP and SCION in SeNDLog and verify path authenticity properties of both protocols.

Keywords: declarative languages, secure routing protocols, verification.

References

1. Cortier, V., Degrieck, J., Delaune, S.: Analysing routing protocols: four nodes topologies are sufficient. In: Proceedings of POST (2012)
2. Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol Composition Logic (PCL). Electronic Notes in Theoretical Computer Science 172, 311–358 (2007)

Translating between equational theories for automated reasoning (Work in progress)

Ben Smyth¹, Myrto Arapinis², and Mark D. Ryan²

¹ INRIA Paris-Rocquencourt, France

² School of Computer Science, University of Birmingham, UK

In the symbolic model of protocol analysis, we typically assume that cryptographic protocols should achieve their objectives in the presence of an adversary that has full control of the network (sometimes called the Dolev-Yao attacker) and cryptography is usually assumed to be perfect. Since the adversary has complete control of the network, messages may be read, modified, deleted, or injected. The adversary is also able to manipulate data contained within those messages, under the restriction of perfect cryptography, that is, the attacker is only able to perform cryptographic operations when in possession of the required keys. It follows, for example, that an adversary may compute the i th element of a tuple or, given the necessary keys, decrypt ciphertexts. The relationships between cryptographic primitives are captured using a set of deduction rules. For example, an equation modelling symmetric decryption can be expressed as $\text{dec}(k, \text{enc}(k, m)) = m$. This equation captures the notion that symmetric encryption is perfect: given a ciphertext $\text{enc}(k, m)$, the plaintext m can only be recovered using the secret key k . If the encryption scheme has some other characteristic (for example, if the scheme is homomorphic), then it is important that these properties are also captured to avoid missing attacks.

Formally, cryptographic primitives are modelled as function symbols and relationships between primitives are captured using an equational theory. Unfortunately, the desired set of function symbols and associated equations may not be amenable to automated analysis, for example, the equational theory may be non-convergent or non-linear, or the set of function symbols may be infinite. Moreover, the algorithms used for analysis may not terminate. Sometimes it is possible to eliminate these problems by abstraction: prove a result using an alternative set of function symbols and associated equations, and prove that this result implies the desired result. In our work, we propose some conditions for translation between function symbols and equational theories such that results transfer in this way.

Our contribution (work in progress). In the context of observational equivalence in the applied pi calculus, we introduce some rules for the translation between function symbols and equational theories. Moreover, we demonstrate that our rules are sound using the intermediate applied pi calculus (a direct proof of soundness would be convoluted since the preservation of structural equivalence is non-trivial). Our results allow us to derive theories which are suited to automated analysis using Blanchet's ProVerif, hence, we facilitate the automatic symbolic analysis of cryptographic protocols.

Using Interpolation for the Verification of Security Protocols (Extended Abstract)*

Giacomo Dalle Vedove, Marco Rocchetto, Luca Viganò, and Marco Volpe

Dipartimento di Informatica, Università di Verona, Italy

Interpolation has been successfully applied in formal methods for model checking and test-case generation for sequential programs. Security protocols, however, exhibit such idiosyncrasies that make them unsuitable to the direct application of such methods, most notably, the fact that, in the presence of a Dolev-Yao intruder, a security protocol is not a sequential program (since the intruder can freely interleave his actions with the normal protocol execution).

We have addressed this problem to develop an interpolation-based method for security protocol verification. Our method starts from a formal protocol specification, along with the specification of a security property (e.g., authentication or secrecy) and a finite number of protocol sessions. It creates a corresponding sequential non-deterministic program, where parameters are introduced in order to capture the behavior of an intruder, in the form of a control flow graph, and then defines a set of goals (representing possible attacks on the protocol) and searches for them by symbolically executing the program. When a goal is reached, an attack trace is extracted from the set of constraints that the execution of the path has produced; such constraints basically represent conditions over parameters that allow one to reconstruct the attack trace(s) found. When the search fails to reach a goal along a given path, a backtrack phase starts, during which the nodes of the graph are annotated (according to the IntraLA algorithm defined by McMillan for sequential programs, which we adapted and slightly modified by omitting unnecessary parameters) with formulas obtained by using Craig interpolation. Such formulas express conditions over the program variables, which, when implied from the program state of a given execution, ensure that no goal will be reached by going forward. The annotations are thus used to guide the search: we can discard a branch when it leads to a state where the annotation of the corresponding node is implied. Summing up, the output of the method is a proof of (bounded) verification in the case when no goal location can be reached starting from a finite-state specification; otherwise, one or more attack traces are produced.

We have begun implementing our method (where we use the Z3 prover, on first-order theories, to model the deductive power of the intruder and calculate the interpolants) and we have applied it to a number of concrete examples. The first results are very promising. We are currently working towards unbounded verification and the automated extraction, from discovered attack traces, of test cases to be applied to security protocol implementations.

* This work was partially supported by the FP7-ICT-2009-5 Project no. 257876, “SPaCioS: Secure Provision and Consumption in the Internet of Services”, and by the PRIN 2010-2011 project “Security Horizons”.

Bounded Memory Protocols and Progressing Collaborative Systems

Max Kanovich¹, Tajana Ban Kirigin², Vivek Nigam³, and Andre Scedrov⁴

¹ Queen Mary, University of London, UK,
mik@dcs.qmul.ac.uk

² University of Rijeka, HR
bank@math.uniri.hr

³ Federal University of Paraíba, João Pessoa, Brazil
vivek@ci.ufpb.br

⁴ University of Pennsylvania, Philadelphia, USA
scedrov@math.upenn.edu

Abstract. It is well-known that the Dolev-Yao adversary is a powerful adversary. Besides acting as the network, intercepting, sending, and composing messages, he can remember as much information as he needs. That is, his memory is unbounded. We recently proposed a weaker Dolev-Yao like adversary, which also acts as the network, but whose memory is bounded. We showed that this Bounded Memory Dolev-Yao adversary, when given enough memory, can carry out many existing protocol anomalies. In particular, the known anomalies arise for *bounded memory protocols*, where there is only a bounded number of concurrent sessions and the honest participants of the protocol cannot remember an unbounded number of facts nor an unbounded number of nonces at a time. This led us to the question of whether it is possible to infer an upper-bound on the memory required by the Dolev-Yao adversary to carry out an anomaly from the memory restrictions of the bounded protocol. This paper answers this question negatively (Theorem 2). The second contribution of this paper is the formalization of Progressing Collaborative Systems that may create fresh values, such as nonces. In this setting there is no unbounded adversary, although bounded memory adversaries may be present. We prove the NP-completeness of the reachability problem for Progressing Collaborative Systems that may create fresh values.

1 Introduction

In the symbolic verification of protocol security, one considers a powerful adversary model now usually referred to as the Dolev-Yao adversary, which arose from positions taken by Needham and Schroeder [18] and a model presented by Dolev and Yao [7]. Not only can the Dolev-Yao adversary act as the network, intercepting, sending and composing messages, but he can also remember as much information as he needs. The goal in protocol verification is to demonstrate that such a powerful adversary cannot discover some secret information, when using some protocol(s). Clearly, if it is shown that such a powerful adversary cannot discover the secret symbolically, then weaker adversaries will also not be able to discover the secret.

In [11], we proposed a Bounded Memory Dolev-Yao adversary, which is very similar to the Dolev-Yao adversary. He also acts as the network, intercepting, sending and composing messages, but differently from the Dolev-Yao adversary, he can remember only a bounded number of facts at a given time. So, in order for him to learn some new information, such as a nonce, he might have to forget some information he previously learned. Clearly, our Bounded Memory Dolev-Yao adversary is weaker than the Dolev-Yao adversary, as the former's memory is bounded, while the latter's is not.

However, despite being weaker, we demonstrated in [11] that many known anomalies can also be carried out by our Bounded Memory Dolev-Yao adversary. We also noticed that the protocols for which we could replay the anomaly with our bounded memory adversary were all *bounded memory protocols*, where one considers that the memory of the system is bounded. That is, in concurrent runs the honest participants of the protocol also cannot remember an unbounded number of facts nor an unbounded number of nonces at a time. This led us to the question of whether it is possible to infer an upper bound on the memory of the Dolev-Yao adversary with respect to the memory restrictions of bounded memory protocols, that is, of the memory used by the participants.

This paper answers this question negatively. That is, it is not possible to determine an upper bound on the memory of the Dolev-Yao adversary even if the memory of the protocol is bounded. From our main result (Theorem 2), we can infer that the Standard Dolev-Yao intruder cannot be constructively approximated by an infinite sequence of increasing memory Bounded Memory Intruders. We show this negative result by proposing a novel undecidability proof for the secrecy problem with the Dolev-Yao adversary. Our undecidability result strengthens the one given in [3, 8], confirming the hardness of protocol verification. In particular, we show that the secrecy problem is “very undecidable:” the secrecy problem is undecidable *even for bounded memory protocols* and thus a bound on the memory of the Dolev-Yao adversary is not computable from a bound on the memory used by a protocol. This is accomplished by a novel encoding of Turing machines by means of memory bounded protocols.

The second contribution of this paper is the formalization of Progressing Collaborative Systems that may create fresh values. We are in particular interested in Collaborative Systems [16] that occur in a closed room, where no other agent can enter and where all agents have bounded memory. We ignore concerns about an outside intruder, although inside adversaries may be present, but have bounded memory. We introduced the notion of progressing in [12] inspired by protocols, namely, by the fact that a protocol session is always progressing. That is, once one step of a protocol session is taken, the same step is no longer repeated. Administrative systems normally also have this progressing nature: once an item in an activity to-do list is checked, that activity is not repeated.

However, in [12], we limited ourselves to systems that did not create fresh values, such as nonces. Combining the progressing condition with the creation of fresh values turned out to be surprisingly challenging because of a subtle interaction between the two features. We discuss this in detail in Section 4. This paper extends the formalization of Progressing in [12] to systems that may create fresh values, based on the machinery

introduced in [11]. We also prove that the reachability problem for Progressing Systems that may create fresh values in NP-complete.

This paper is structured as follows:

- Section 2 reviews the specification of bounded memory protocols, the Dolev-Yao Adversary, and of Bounded Memory Adversaries. It also reviews some of the complexity results for the secrecy problem;
- Section 3 contains the secrecy undecidability proof with memory bounded protocols. This is a novel, stronger undecidability proof, which allows us to infer that it is not possible to determine an upper bound on the memory of the Dolev-Yao adversary from the memory bound of the protocol;
- Section 4 contains the formalization of Progressing Collaborative System that may create nonces. We argue that its precise formalization is only meaningful when bounding the memory of the participants of the system. We also prove the NP-completeness of the reachability problem;
- Finally in Sections 5 and 6 we comment on related work and conclude by pointing out to future work.

2 Bounded Memory Protocols and Adversaries

We formalize bounded memory protocol theories and adversary theories by means of multiset rewrite rules, similarly as in [3, 8]. A set of rewrite rules, or a theory, was proposed in [3, 8] for modeling protocols and the standard Dolev-Yao intruder with unbounded memory. In order to carefully compare our complexity results, we closely follow this approach and adapt the theories from [3, 8] to formalize bounded memory protocols and Bounded Memory Adversaries.

Assume fixed a sorted first-order alphabet consisting of constant symbols, c_1, c_2, \dots , function symbols, f_1, f_2, \dots , and predicate symbols, P_1, P_2, \dots all with specific sorts (or types). The multi-sorted terms over the signature are expressions formed by applying functions to arguments of the correct sort. A *fact* is a ground, atomic formula over multi-sorted terms. Facts have the form $P(t_1, \dots, t_n)$ where P is an n -ary predicate symbol, where t_1, \dots, t_n are terms, each with its own sort.

The *size of a fact* is the total number of term and predicate symbols it contains. We count one for each predicate, function, constant, and variable symbols. We use $|F|$ to denote the size of a fact F . For example, $|P(x, c)| = 3$, and $|P(f(z, x, n), z)| = 6$. We will normally assume in this paper an upper bound on the size of facts, as in [3, 8, 16].

A *state*, or *configuration* of the system is a finite multiset of grounded facts, *i.e.*, facts that do not contain variables. Configurations, intuitively, specify the state of the world and are modified by actions. In general, an action is a multiset rewrite rule of the following form:

$$X_1, \dots, X_n \longrightarrow \exists \mathbf{x}. Y_1, \dots, Y_m \quad (1)$$

where the X_i s and Y_j s are facts. The collection X_1, \dots, X_n is called the pre-condition of the rule, while Y_1, \dots, Y_m is called post-condition. We assume that all free variables are universally quantified. By applying the action for a ground substitution (σ),

the pre-condition applied to this substitution $(X_1\sigma, \dots, X_n\sigma)$ is replaced with the post-condition applied to the same substitution $(Y_1\sigma, \dots, Y_m\sigma)$. In this process, the existentially quantified variables (x) appearing in the post-condition are replaced by fresh constants, also called nonces in protocol security literature. The rest of the configuration remains untouched. Thus, we can apply the action $P(x), Q(y) \rightarrow_A \exists z. R(x, z), Q(y)$ to the global configuration $V, P(t), Q(s)$ to get the global configuration $V, R(t, c), Q(s)$, where the constant c is new.

Given a multiset rewrite system R , one is often interested in the *reachability problem*: Is there a sequence of (0 or more) rules from R which transforms configuration W into Z ? If this is the case then we say that Z is reachable from W using R .

Balanced Actions and Empty Facts An important condition for formalizing bounded protocols is that of *balanced actions*. Balanced actions were introduced in the context of collaborative systems [16]. We classify an action as *balanced* if the number of facts in its pre-condition is the same as the number of facts in its post-condition. That is, $n = m$ in Equation 1. If we restrict all actions in a system to be balanced, then the size of all configurations in a run remains the same as in the initial configuration. Since we assume facts to have a bounded size, the use of balanced actions imposes a bound on the storage capacity of the agents, *i.e.*, balanced systems have constant memory. Creating a new fact by means of a balanced action amounts to inserting that fact into the resulting configuration by replacing a fact appearing in the enabling configuration. In other words the memory of the system is only updated. No new memory space is created.

In order to support the creation of new facts in balanced systems, we use *empty facts*, written $P(*)$. Intuitively, an empty fact denotes an available memory slot that could be filled by some new information. Here $*$ is not a constant, but just used for illustrative purposes. By using empty facts, one can transform unbalanced systems into balanced systems simply by adding enough empty facts to the pre-condition or the post-condition of each rule with so that it becomes balanced. The obtained balanced system can be considered as equivalent to the original, unbalanced one, provided there is no bound on the size of configurations.

2.1 Bounded Memory Protocols

A bounded memory protocol, formally defined below, only contains balanced actions [11]. This means that the number of facts known by the participants at a given time is bounded. Bounding the memory available for protocol sessions also intuitively bounds the number of concurrent protocol sessions. This is because for each protocol session, one needs some free memory slots to remember, for instance, the internal states of the agents involved in the session. However, this does not mean that there may not be an unbounded number of protocol sessions in a trace. Once a protocol session is completed, the memory slots it required can be re-used to initiate a new protocol session.

This is different to the well-founded protocol theories in [3, 8] where the rules are not necessarily balanced and where all protocol sessions are created at the beginning of the trace before any protocol session starts executing. In well-founded protocol theories, an unbounded number of protocol sessions can run concurrently and therefore participants are allowed to remember an unbounded number of facts.

Definition 1. A theory \mathcal{A} is a balanced role theory if there is a finite list of predicate names called the role states S_0, S_1, \dots, S_m for some m , such that every rule $L \rightarrow \exists t.R$ in \mathcal{A} is balanced and there is exactly one occurrence of a state predicate in L , say S_i , and exactly one occurrence of a state predicate in R , say S_j , such that $i < j$. We call the first role state, S_0 , initial role state, and the last role state S_m final role state. Only rules with final role states can have an empty fact in the post-condition.

Defining roles in this way, ensures that each application of a rule in \mathcal{A} advances the state forward. Each instance of a role can only result in a finite number of steps in a trace. The request on empty facts formalizes the fact that one of the participants, either the initiator or the responder, sends the “last” protocol message. In [11], one can find several examples of protocols specified as balanced role theories.

In order to allow an unbounded number of protocol sessions in a trace, we allow protocol roles to be created at any time with the cost of consuming empty facts $P(*)$. At the same time, we allow protocol sessions that have been completed to be forgotten. Once a final role state has been reached, it can be deleted, creating new empty facts $P(*)$ in the process. These empty facts can then be used to create new protocol roles starting hence a new protocol session. Such theories are called role regeneration theories.

Definition 2. If $\mathcal{A}_1, \dots, \mathcal{A}_k$ are balanced role theories, a role regeneration theory is a set of rules that either have the form

$$Q_1(x_1) \cdots Q_n(x_n) P(*) \rightarrow Q_1(x_1) \cdots Q_n(x_n) S_0(x),$$

where $Q_1(x_1) \cdots Q_n(x_n)$ is a finite list of facts not involving any role states, and S_0 is the initial role state for one of theories $\mathcal{A}_1, \dots, \mathcal{A}_k$, or the form

$$S_m \rightarrow P(*), \text{ where } S_m \text{ is the final state for one of theories } \mathcal{A}_1, \dots, \mathcal{A}_k.$$

This definition is a central difference to the setting in [3, 8]. In [3, 8] one assumed that all protocol sessions are initialized at the beginning of the trace, that is, all protocol sessions run concurrently. This means that there is no bound on the memory of the (honest) participants since they need to remember that they participate in a possibly unbounded number of protocol sessions. Under the definition above, on the other hand, this is no longer the case as the explicit use of balanced actions in role theories and role regeneration theories allows us to bound the memory of the participants, including the number of concurrent protocols in the system, without bounding the total number of sessions in a trace.

Definition 3. A pair (\mathcal{P}, H) is a bounded memory protocol theory if H is a finite set of facts (called initial set), and $\mathcal{P} = \mathcal{R} \uplus \mathcal{A}_1 \uplus \dots \uplus \mathcal{A}_n$ is a protocol theory where \mathcal{R} is a role regeneration theory involving only facts from H and the initial and final roles states of $\mathcal{A}_1, \dots, \mathcal{A}_n$, and $\mathcal{A}_1, \dots, \mathcal{A}_n$ are balanced role theories. For role theories \mathcal{A}_i and \mathcal{A}_j , with $i \neq j$, no role state predicate that occurs in \mathcal{A}_i can occur in \mathcal{A}_j .

Intuitively, a bounded memory protocol theory specifies a particular scenario to be model-checked involving some given protocol(s). Besides empty facts, $P(*)$, the finite initial set of facts contains all the facts with the information necessary to start protocol sessions, for instance, shared and private keys, the names of the participants of the network, as well as any compromised keys. Here, for simplicity, we assume only symmetric keys, although other types of keys can be also formalized.

I/O Rules: REC : $N_S(x) \rightarrow M(x)$ SND : $M(x) \rightarrow N_R(x)$	I/O Rules: REC: $N_S(x) \rightarrow M(x)$ SND: $M(x) \rightarrow N_R(x)$
Decomposition Rules: DCMP : $M(\langle x, y \rangle) \rightarrow M(x) M(y)$ DECS : $M(k) M(enc(k, x)) \rightarrow$ $M(k) M(enc(k, x)) M(x)$	Decomposition Rules: DCMP: $M(\langle x, y \rangle) P(*) \rightarrow M(x) M(y)$ DEC: $M(k) M(enc(k, x)) P(*)$ $\rightarrow M(k) M(x) M(enc(k, x))$
Composition Rules: COMP : $M(x) M(y) \rightarrow M(\langle x, y \rangle)$ USE : $M(x) \rightarrow M(x) M(x)$ ENCS : $M(k) M(x) \rightarrow$ $M(k) M(enc(k, x))$ GEN : $\rightarrow \exists n. M(n)$	Composition Rules: COMP: $M(x) M(y) \rightarrow M(\langle x, y \rangle) P(*)$ USE: $M(x) P(*) \rightarrow M(x) M(x)$ ENC: $M(k) M(x) \rightarrow M(k) M(enc(k, x))$ GEN: $P(*) \rightarrow \exists n. M(n)$
	Memory maintenance rule: DELM: $M(x) \rightarrow P(*)$

a. Theory for the Standard Dolev-Yao Adversary b. Bounded Memory Dolev-Yao Adversary Theory

Fig. 2: Theories for the Standard and the Bounded Memory Adversaries

2.2 Standard Dolev-Yao and Bounded Memory Dolev-Yao Adversaries

The powerful adversary proposed by Dolev-Yao [7] acts as the network, that is, all messages communicated are sent through the adversary. He hears everything and learns messages modulo encryption. More precisely, he is capable of intercepting any message sent by a protocol participant and then store the received information, decompose it and decrypt with the keys he possesses. He cannot, however, decrypt messages for which he does not have the correct key. Moreover, he can also create fresh values, encrypt, compose messages from the information he has learned. One of his major strengths is that he can remember as much information as he wants, *i.e.*, his memory is unbounded.

Figure 1a. depicts the rules of such an adversary. The I/O rules specify the fact that the adversary acts as the network receiving all messages sent (N_S) and sending all messages that are received (N_R). The remaining rules are straightforward, specifying when the adversary may decompose and compose messages. Notice that contrary to the formalization of the bounded memory protocols, the actions specifying the Dolev-Yao adversary are not all balanced. In particular, the adversary may always learn new facts, such as in the actions DECS and GEN, where the adversary learns the contents of an encrypted message and creates a nonce.

In [11], we proposed a Bounded Memory Dolev-Yao adversary, which has many capabilities of the Dolev-Yao adversary. He can intercept, send and compose messages, create nonces, etc. But differently from the Dolev-Yao adversary, he can remember only a bounded number of facts of a bounded size, at any given time. This is formally imposed by the balanced adversary theory presented in Figure 1b. In order for him to store some new information, such as a nonce, he might have to forget some information he previously learned. This is specified by additional memory maintenance rule.

2.3 Complexity Results for the Secrecy Problem

In an interaction of malicious adversaries with honest participants, one is interested in *secrecy problem*, namely, in determining whether the adversary can discover a secret s . Formally it is an instance of the reachability problem: Is it the case that a configuration containing $M(s)$, where s is a secret originally owned by an honest participant can be reached from an initial configuration?

Undecidability of the Secrecy Problem It is known for some time that the secrecy problem is undecidable in general [3, 8]. The undecidability proof in [3, 8] proceeds by encoding the existential Horn implication problem, which is also proved to be undecidable. However, in that work, one used *well-founded protocol theories*, where the memory of the protocol is unbounded. For instance, in *well-founded protocol theories*, it is allowed for an unbounded number of concurrent protocol sessions to run at the same time. In fact, all the protocol sessions in a trace are initialized at the beginning before any session starts. This implies that the participants of the system may remember an unbounded number of facts, namely, the facts containing the information of the protocols in which they are participating in.

In Section 3, we strengthen the result in [3, 8], by showing that the secrecy problem is undecidable *even if the memory of the protocol is bounded*. This is accomplished by a novel encoding of Turing machines by means of memory bounded protocols.

PSPACE-completeness of the Secrecy problem for the Bounded Memory Dolev-Yao Adversary Besides proposing the Bounded Memory Dolev-Yao Adversary and demonstrating that he can carry out known anomalies when given enough memory, we proved in [11] that the secrecy problem when assuming the Bounded Memory Dolev-Yao Adversary is PSPACE-complete. The key insight for this result was showing how to handle the fact that a trace may have an exponential number of nonces, which seems to preclude PSPACE membership. We circumvent the problem of requiring too many fresh values in a trace by reusing obsolete constants instead of creating new values.

The argument goes roughly like this: we assume a balanced system that consists of a number of honest participants and a Bounded Memory Dolev-Yao Adversary and an upper bound on the size of all facts. Since all actions of the system are balanced, including those specifying the adversary (see Figure 1b), the number of facts in any configuration remains the same as in the initial configuration, namely m . Moreover, as we assume an upper bound on the size of facts, namely k , then any configuration in a trace has at most mk symbols. We can then fix a priori a polynomial number of nonce names, namely, $2mk$ names, so that whenever one needs a fresh nonce, one can find a name in this set of $2mk$ names that is fresh to the participants. It may well be the case that some name in this set of nonce names is used many times in a trace. However, for the participants at that point of the trace, the name used seems fresh as no participant can remember it.

This idea will be key for our proposal of Progressing systems with nonce generation in Section 4.

3 Protocol security is very undecidable: A bound on the adversary cannot be inferred from a bound on a protocol

We now detail the sound and faithful encoding of Turing machines using bounded memory protocols. We show that an attack on the given protocol by *an unbounded, standard Dolev-Yao intruder* is possible if and only if the encoded Turing machine terminates. From that we infer the undecidability of a Dolev-Yao attack *even for bounded memory protocols*. Notice that our result works even if we assume a (large enough) bound on the size of facts, *e.g.*, a bound a bit greater than 30.

3.1 Encoding of Turing Machine Tapes

Without loss of generality, let \mathcal{M} be a Turing machine such that

- (i) \mathcal{M} has only one tape, which is one-way unbounded to the right. The leftmost cell (numbered by 0) contains the marker \$ unerased;
- (ii) The initial 3-cell configuration is of the following form, where B stands for the blank symbol:

$$\boxed{\$} \mid \boxed{\langle q_1, B \rangle} \mid \boxed{B} \quad (2)$$

We write $\boxed{\langle q, \xi \rangle}$ to denote that the corresponding cell contains the symbol ξ and is scanned by \mathcal{M} in its state q .

- (iii) We assume that all instructions are “move” instructions. The head of \mathcal{M} cannot move to the leftmost cell marked with \$.
- (iv) Finally, \mathcal{M} has only one *accepting* state, q_0 .

Encoding of the Tape In our encoding, we need two honest participants only, Alice and Bob. Assume they share a symmetric key K , not known to any other participant. We will encode the tape cells separately as follows:

- (a) An unscanned cell that contains symbol ξ_0 is encoded by a term encrypted with the key K
 $E_K(\langle t_0, \xi_0, e_0, t_1 \rangle)$,
 where t_0 and t_1 are nonces, and $e_0 = 1$ if the cell is the last cell in a configuration.
- (b) The cell that contains symbol ξ and is scanned by \mathcal{M} in state q is also encoded by a term encrypted with the key K
 $E_K(\langle t_1, \langle q, \xi \rangle, 0, t_2 \rangle)$
 where t_1 and t_2 are nonces.

Motivation: The nonces t_0 and t_1 in the terms encoding the tape cell are used for two purposes:

- (a) Firstly, t_0 and t_1 serve as “timestamps” for the visit by \mathcal{M} in the cell. Whenever \mathcal{M} re-visits this cell, the previous term is updated with fresh nonces indicating a new visit;
- (b) Secondly, as t_0 and t_1 are unique, they are used to uniquely link cells that are adjacent to each other.

For example, the initial configuration, Equation (2), with three cells is encoded by using the sequence of nonces t_0, t_1, t_2, t_3 as shown below:

$$\langle E_K(\langle t_0, \$, 0, t_1 \rangle), E_K(\langle t_1, \langle q_1, B \rangle, 0, t_2 \rangle), E_K(\langle t_2, B, 1, t_3 \rangle) \rangle$$

Notice the role of the nonces t_0, t_1, t_2, t_3 . For instance, the nonce t_1 is used to correctly encode the fact that the cell $\langle q_1, B \rangle$ is to the right of the cell with the mark \$.

3.2 Encoding Turing Machine's Actions as a Bounded Memory Protocol

Given a Turing machine \mathcal{M} and the encoding of tapes discussed above, we encode its actions by means of bounded memory protocol called $\mathcal{P}_{\mathcal{M}}$. We describe the role of Alice (initiator) and Bob (responder):

Alice's Role Assume that Alice is the initiator and her initial state is:

$$\langle E_K(\langle t_0, \$, 0, t_1 \rangle), E_K(\langle t_1, \langle q, B \rangle, 0, t_2 \rangle), E_K(\langle t_2, B, 1, t_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

The protocol starts by Alice updating all nonces t_i to t'_i , and sending the following updated message to Bob. At this point, she does not need to remember the previous terms using the nonces t_i . Notice that the last term does not share nonces with the first three. It will be used for extending the tape.

$$\langle E_K(\langle t'_0, \$, 0, t'_1 \rangle), E_K(\langle t'_1, \langle q, B \rangle, 0, t'_2 \rangle), E_K(\langle t'_2, B, 1, t'_3 \rangle), E_K(\langle t'_4, B, 1, t'_5 \rangle) \rangle$$

That is she erases her memory and is ready to store new facts. In particular, she is waiting for a message from Bob of the form:

$$\langle E_K(\langle t_0, \alpha_0, 0, t_1 \rangle), E_K(\langle \tilde{t}_1, \alpha_1, 0, \tilde{t}_2 \rangle), E_K(\langle t_2, \alpha_2, e_2, t_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

By verifying its integrity with $(t_1 = \tilde{t}_1)$ and $(t_2 = t_2)$, Alice assumes that there is no intrusion in the channel. If some α_i is of the form $\langle q_0, \xi \rangle$, then Alice sends openly a *secret* to Bob, otherwise, Alice sends a neutral message.

Bob's role The role of Bob is to transform the message received with the help of an instruction from the given Turing machine \mathcal{M} . Bob is expecting to receive a message (presumably from Alice) of the form:

$$\langle E_K(\langle t_0, \xi_0, 0, t_1 \rangle), E_K(\langle \tilde{t}_1, \langle q, \xi \rangle, 0, \tilde{t}_2 \rangle), E_K(\langle t_2, \xi_2, e_2, t_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

Bob verifies its integrity by $(t_1 = \tilde{t}_1)$ and $(t_2 = t_2)$, and follows one of three cases:

(1) (Extending the tape) For $e_2 = 1$, Bob *updates* nonces t_i to t'_i , and sends the following updated message to Alice, which provides a new last cell in the chain of four cells

$$\langle E_K(\langle t_0, \xi_0, 0, t'_1 \rangle), E_K(\langle t'_1, \langle q, \xi \rangle, 0, t'_2 \rangle), E_K(\langle t'_2, \xi_2, 0, t'_3 \rangle), E_K(\langle t'_3, B, 1, t'_4 \rangle) \rangle$$

(2) (Moving the Head of the Machine to the Right) For an \mathcal{M} 's instruction of the form $q\xi \rightarrow q'\eta R$, denoting: "if in state q looking at symbol ξ , replace it by η , move the tape head one cell to the right, and go into state q' ", Bob *updates* some nonces t_i to t'_i , and sends the following updated message to Alice

$$\langle E_K(\langle t_0, \xi_0, 0, t'_1 \rangle), E_K(\langle t'_1, \eta, 0, t'_2 \rangle), E_K(\langle t'_2, \langle q', \xi_2 \rangle, 0, t'_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

(3) (Moving the Head of the Machine to the Left) For an \mathcal{M} 's instruction of the form $q\xi \rightarrow q'\eta L$, denoting: "if in state q looking at symbol ξ , replace it by η , move the tape head one cell to the left, and go into state q' ", Bob *updates* some nonces t_i to t'_i , and sends the following updated message to Alice

$$\langle E_K(\langle t_0, \langle q', \xi_0 \rangle, 0, t'_1 \rangle), E_K(\langle t'_1, \eta, 0, t'_2 \rangle), E_K(\langle t'_2, \xi_2, 0, t'_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

Remark 1. Both Alice and Bob can input and output only messages of the form

$$\langle E_K(\langle t_0, \alpha_0, 0, t_1 \rangle), E_K(\langle t_1, \alpha_1, 0, t_2 \rangle), E_K(\langle t_2, \alpha_2, e_2, t_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

where the first three components represent the chain of three cells, and the fourth component refers to the last cell in a configuration.

Remark 2. The above protocol is balanced. It can be formalized by a bounded memory protocol see [13]. In particular, only terms of height fixed in advance are used. Nonces are only updated, that is, the old nonces are replaced by new nonces. Therefore, Alice and Bob can forget the old nonces. In fact, Alice and Bob are finite automata, which are allowed to *update nonces* only.

3.3 A Man-in-the-Middle Attack by Mallory

Notice that, according to Remark 1, by active eavesdropping Mallory can accumulate terms of the form

$$E_K(\langle t_1, \alpha_1, e_1, t_2 \rangle) \quad (3)$$

if and only if they are components of outputs generated by Alice or by Bob. We now discuss the following attack on the protocol above:

(1) For the first run, Mallory intercepts the initial message from Alice, stores it, and resends it to Bob. While Bob responds, Mallory intercepts the message from Bob, stores it, and resends it to Alice.

(2) For each of the next runs, Mallory first intercepts the initial message from Alice. Taking non-deterministically terms of the form (3) from his memory, Mallory then composes a message of the form:

$$\langle E_K(\langle t_0, \alpha_0, 0, t_1 \rangle), E_K(\langle \tilde{t}_1, \alpha_1, 0, \tilde{t}_2 \rangle), E_K(\langle t_2, \alpha_2, e_2, t_3 \rangle), E_K(\langle t_4, B, 1, t_5 \rangle) \rangle$$

and sends it to Bob. If Bob accepts this message and responds with a transformed one as described in the protocol, then Mallory intercepts this new message from Bob, stores it, and resends it to Alice.

The following lemma shows a certain chain-like structure of the terms accumulated by the adversary. These chain-like structure are specified by the use of nonces and each chain corresponds to reachable configurations of the Machine \mathcal{M} .

Lemma 1. *Suppose that a term of the form $E_K(\langle t, \langle q, \xi \rangle, 0, t' \rangle)$ appears in the intruder memory by active eavesdropping. Then there is a unique sequence of nonces t_0, t_1, \dots, t_{n+2} and a chain of terms from the adversary's memory*

$$\begin{aligned} & E_K(\langle t_0, \$, 0, t_1 \rangle), E_K(\langle t_1, x_1, 0, t_2 \rangle), \dots, E_K(\langle t_{j-1}, x_{j-1}, 0, t_j \rangle), \\ & E_K(\langle t_j, \langle q, x_j \rangle, 0, t_{j+1} \rangle), E_K(\langle t_{j+1}, x_{j+1}, 0, t_{j+2} \rangle), \dots, E_K(\langle t_n, x_n, 0, t_{n+1} \rangle), \\ & E_K(\langle t_{n+1}, B, 1, t_{n+2} \rangle) \end{aligned}$$

such that

- (a) $t_j = t$, $x_j = \xi$, and $t_{j+1} = t'$,
- (b) \mathcal{M} leads from the empty initial configuration to the configuration where the string $x_1 x_2 \dots x_j \dots x_n$, is written in cells $1, 2, \dots, j, \dots, n$ on the tape

\$	x_1	x_2	\cdot	\cdot	x_j	\cdot	\cdot	x_n		\dots
----	-------	-------	---------	---------	-------	---------	---------	-------	--	---------

and the j -th cell is scanned by \mathcal{M} in state q .

Proof. By induction on the number of actions performed by Bob to outcome a message one of the components of which is $E_K(\langle t, \langle q, \xi \rangle, 0, t' \rangle)$. ■

Theorem 1. *There is a Dolev-Yao attack on the above protocol if and only if the machine \mathcal{M} terminates on the empty input.*

Proof. We sketch the proof of both directions of the proof.

- (a) The direction from a terminating computation to an attack is straightforward by induction on the length of the computation.
- (b) The inverse direction is quite tricky. In the case of a successful attack, a term of the form $E_K(\langle \tilde{t}_1, \langle q_0, \xi \rangle, 0, \tilde{t}_2 \rangle)$, must appear in the adversary's memory. Then by Lemma 1, \mathcal{M} leads from the empty initial configuration to a final configuration where a cell is scanned in state q_0 . ■

Notice that in all attacks above the attacker in fact does not need to create/update fresh nonces, but simply intercept, decompose, compose and copy messages.

Corollary 1. *The existence of a Dolev-Yao attack is undecidable even for bounded memory protocols, $\mathcal{P}_{\mathcal{M}}$, where Alice and Bob are finite automata whom are allowed to update nonces only, all actions by Alice and Bob are balanced, and only terms of height fixed in advance are used by Alice, Bob, and an adversary (even if the actions of the adversary are limited to decompose, compose, and copy).*

Proof. Given a non-recursive recursively enumerable set S , and a sequence of Turing machines \mathcal{M}_n such that \mathcal{M}_n terminates on the empty input iff $n \in S$, it suffices to consider the corresponding bounded memory protocols $\mathcal{P}_{\mathcal{M}_n}$. ■

Thus an upper bound on the memory of the Dolev-Yao adversary is not computable from a bound on the memory used by a protocol. Based on peculiarities of our encoding described in Section 3.2, we can express such a phenomenon in quantitative terms.

Theorem 2. *Whatever a total recursive function h we take, we can construct a recursive sequence of bounded memory protocols \mathcal{Q}_n so that*

- (a) *For any n , there is a Dolev-Yao attack on the bounded memory protocol \mathcal{Q}_n .*
- (b) *However, for any n starting from some n_0 , any Dolev-Yao adversary the size of whose memory is bounded by $h(n)$ is not capable of detecting an attack on the bounded memory protocol \mathcal{Q}_n .*

Proof Sketch. Given a total recursive function f , as \mathcal{Q}_n we take the bounded memory protocol $\mathcal{P}_{\mathcal{M}_n}$ described in Section 3.2, where \mathcal{M}_n is a Turing machine terminating on the empty input with the value $f(n)$.

Roughly, according to Theorem 1, Mallory, whose memory size is bounded by $h(n)$, can play at most $2^{O(h(n))}$ steps. It suffices, therefore, to take the function f such that its time complexity is $\Omega(2^{2^{h(n)}})$. ■

The Theorem above implies that the Standard Dolev-Yao intruder cannot be constructively approximated by an infinite sequence of increasing memory Bounded Memory Intruders.

4 Progressing Collaborative Systems with Fresh Values

We introduced the notion of progressing in [12] in the context of Collaborative Systems where agents interact in a closed-room setting, and no outside intruder is present. Nevertheless, there may be adversaries inside the system. We are in particular interested in systems where all agents have bounded memory, even the inside adversaries. Collaborative systems can be modelled with multiset rewriting, for instance the multiset rewriting rules for the bounded memory intruder that may be present in the system is shown in Figure 1b.

Progressing is inspired by the nature of security protocols, as well as many administrative and business processes. Namely, once one step of a protocol session is taken, the same step is not repeated. Similarly, whenever one initiates some administrative task, one receives a “to-do” list with the activities or tasks that have to be performed or achieved. Once an item on the list has been “checked”, one does not need to return to this item anymore. When all the items have been checked, the process ends. Such a process is always advancing and it is completed within a bounded number of transactions. Additionally, such processes often manipulate a bounded number of values. Consider, for example, the simple process where a bank customer needs a new PIN number: The bank will assign the customer a new PIN number, which is often a four digit number and hence bounded. Even when a customer is allowed to chose a PIN number or some password, it has to satisfy some conditions, e.g., all its characters must be alphanumeric and, in practice, the password is bounded since users are never able to use an unbounded password due to buffer sizes, etc. Consequently, protocols and administrative processes have a polynomial number of steps, in other words they can be considered as *efficient*. That is, one does not need to perform an exponential number of actions to conclude such processes.

To formally capture this intuition, we defined Progressing in [12] as follows: A sequence of actions is *progressing* if *an instance* of an action appears at most once. Here no nonces were allowed, and an instance of an action is obtained by a substitution which replaces all variables appearing in the pre- and post-condition of the action with constants. Assuming a finite signature, *i.e.* a finite number of constant symbols, there is a finite number of instances of any action. This notion of progressing reflects the requirement that progressing processes are efficient, as one needs to consider only traces of polynomial length to check whether a process can be completed or not. For instance, the Towers of Hanoi problem has no progressing plans, since any solution is of exponential length, which implies that one and the same action is necessarily used an exponential number of times. In [12] we show that the progressing reachability problem for systems that do not create nonces is NP-complete.

However, extending this notion of progressing to systems that can create nonces turned out to be quite challenging. The problem arises from the fact that if we allow actions to create fresh values, one may capture processes which require an exponential number of actions, that is, processes that cannot be efficiently carried out. Let us try to extend *naively* the progressing definition above to the case when actions may create nonces as follows: A sequence of actions that may create fresh values is progressing if an instance of an action, with the same constants and the same nonces, appears at most once. Unfortunately, such a definition of progressing is not satisfactory. When a nonce

is created, it is fresh, meaning that it hasn't appeared in the system as yet. Consequently, every application of an action that creates a nonce is a new instance of that action. For instance, we can adapt the encoding of the Towers of Hanoi, so that each move creates a new nonce. Thus each action is a different instance, since a different nonce is used and created. Therefore, according to the above naive definition, the Towers of Hanoi would be progressing, which is clearly not what we want.

Therefore, in order to extend the notion of progressing to the case where actions may create nonces, we shouldn't allow unbounded nonce generation. Instead we need to somehow limit the use of nonces, but how many nonces is enough? This question is answered for the case when systems are balanced. As discussed in Section 2.3, for the case of *balanced systems*: one can simulate any plan that uses an unbounded number of nonces by fixing a priori a polynomial number of nonce names [11] with respect to the number of facts in the initial configuration (m) and the upper-bound on the size of facts (k). In the following sections, we formalize these intuitions.

4.1 Balanced Progressing with Fresh Values

We extend the notion of progressing for balanced systems that can create fresh values. Central to our notion will be the definition of when two instances of actions are equivalent (Definition 4). Consider for example the following two instances of an action, where the t_i s are terms and n_j s are nonce names which do not appear in the alphabet of the language:

$$\begin{aligned} X_1(t_1)X_2(t_2, t_3, n_1)X_3(n_1, n_2) &\rightarrow \exists x. X_4(t_1)X_2(t_2, x, n_3)X_5(n_1, n_3), \\ X_1(t_1)X_2(t_2, t_3, n_4)X_3(n_4, n_5) &\rightarrow \exists x. X_4(t_1)X_2(t_2, x, n_6)X_5(n_4, n_6). \end{aligned}$$

These instances only differ in the nonce names used: the same fresh value, n_3 in the former instance and n_6 in the latter, appear in same facts exactly at the same places, and similarly, for the pairs of nonces (n_1, n_4) , and (n_2, n_5) . Inspired by a similar notion in λ -calculus [4], and α -equivalence among configurations in [11], we regard instances of actions that differ only in the nonce's names used, as equivalent.

Definition 4. Two instances of an action, r_1 and r_2 , are equivalent if there is a bijection σ that maps the set of all nonce names appearing in one instance to the set of all nonce names appearing in the other instance, such that $r_1\sigma = r_2$.

The two instances given above are equivalent because of the following bijection $\{(n_1, n_4), (n_2, n_5), (n_3, n_6)\}$. It is easy to show that the above relation among instances of actions is indeed an equivalence relation.

Definition 5. Given a balanced multiset rewrite system R , an initial configuration W and a polynomial $f(m, k)$, we say that a sequence of actions is progressing if it contains at most $f(m, k)$ equivalent instances of any action, where m is the number of facts in the configuration W and k is the upper bound on size of facts.

Progressing reachability problem has a solution if for a given multiset rewrite system R and configurations W and Z , there is a *progressing* sequence of actions from R which transforms configuration W into Z .

Notice that our new notion of progressing extends progressing from [12], as they coincide when systems do not allow fresh values. We will, therefore, be able to compare our complexity results.

Furthermore, as per Definition 5, not every computation could be considered as progressing. Here a nonce name may only be used by the same action a polynomial number of times in a computation. Hence, not every reachability problem that has a solution will have a progressing solution. For example, in any solution of Towers of Hanoi puzzle, one and the same nonce name has to be updated an exponential number of times by the only action from the representation of this puzzle in [11]. Therefore this problem has no progressing solution as per our Definition 5.

Notice that, if nonces are allowed, we only conceive progressing in balanced systems, while progressing with no nonces is clear for any multiset rewriting system, even the unbalanced ones. This is because nonce update from [11] was only possible in balanced systems.

The progressing reachability problem for balanced systems is NP-complete, as stated by the theorem below. Its proof can be found in [13]. Here we assume that actions of the system, and goal configurations are recognizable in polynomial time.

Theorem 3. *Given a multiset rewrite system R over a finite signature, with only balanced actions that can create fresh values, an initial and a final configuration, an upper-bound, k , on the size of facts and a polynomial f with two parameters, the progressing reachability problem is NP-complete.*

5 Related Work

This paper strengthens the undecidability proof given in [3, 8]. In particular, the proof in [3, 8] uses an encoding with well-founded protocols theories, whereas our proof uses an encoding with bounded memory protocols. While in bounded memory protocols the memory of the honest participants is bounded, in well-founded protocols it is possible for the honest participants to have an unbounded memory. This is in fact the case in the undecidability proof given in [3, 8]. The proof relies on an unbounded number of protocol sessions. Moreover, all these protocols sessions are created before any sessions starts executing, hence participants require an unbounded memory to remember in which protocol sessions they are participating. On the other hand, in our proof, Alice and Bob participate in one protocol session at a time. Whenever one is finished, they can re-use their memory to participate in the subsequent protocol session. This difference is crucial, as with our proof, we can infer that there is no way to compute an upper bound on the memory of the adversary from the memory bounds of the participants, demonstrating further the hardness of the secrecy problem.

Our NP upper bound for the progressing reachability problem in Theorem 3 is different from the NP upper bound obtained [1, 20] in the context of protocol security. In their models, the progressing condition is incorporated syntactically into the rules of the theories. Specifically, they use role predicates of the form A_i contain an index i denoting the stage in the protocol. The NP-completeness result in [1, 20] is obtained by further restricting systems to have only a bounded number of roles. We, on the other

hand, bound the number of instances of actions that can appear in a plan. It would be interesting, however, to check whether our assumption on the existence of an upper-bound on the size of facts could be relaxed as in [20].

Harrison *et al.* present a formal approach to access control [10] and faithfully encode a Turing machine in their system. However, in contrast to our encoding, they use a non-commutative matrix to encode the sequential, non-commutative tape of a Turing machine. In their proofs, the non-commutative nature of the encoding plays an important role. We, on the other hand, encode Turing machine tapes by using commutative multisets. Specifically, they show that if no restrictions are imposed to the systems, the reachability problem is undecidable.

Much work on reachability related problems has been done within the Petri nets community, see *e.g.*, [9]. Specifically, we are interested in the *coverability problem* which is closely related to the reachability problem in multiset rewrite systems. To the best of our knowledge, no work that captures exactly the balanced condition nor the progressing with nonce creation has yet been proposed. In these cases, it does not seem possible to provide direct, *faithful* reductions between our systems and Petri nets.

6 Conclusions

This paper showed that the memory of the adversary cannot be inferred from the memory bounds of the participants (Theorem 2). This is accomplished by proposing a novel undecidability proof by encoding Turing machines by means of bounded memory protocols. This result confirms the hardness of protocol security. It answers negatively an open problem left in [11]. Our second contribution was the formalization of progressing for balanced systems that can create fresh values. We believe that this fragment will provide foundations for a useful class of systems, namely for systems such as administrative processes where the same instance of an action should not be performed an exponential number of times. Finally, we proved the NP-completeness of the Progressing reachability problem.

There are many directions to investigate from here. For instance, it would be interesting to check whether one can adapt the encoding of the Horn implication problem given in [3, 8] to use bounded memory protocols, instead of well-founded ones. Another direction is whether one can improve the NP-completeness proof by relaxing the assumption on the upper-bound of facts. This was possible in the context of protocol security as shown in [20].

We are currently collaborating with Carolyn Talcott on the use of the computational tool Maude [5] for the specification and model-checking of regulated processes, such as administrative processes [15]. In particular, we are investigating whether our NP-completeness proof can improve Maude's performance in model-checking Progressing systems.

Another direction that we are currently investigating is to extend our model with real times. In particular, systems that can create fresh values and mention real times are of great interest to protocol security. For instance, many distance authentication protocols [17, 2] rely on timing measures. Thus extending our model with real times and determining decidable fragments, *e.g.*, balanced systems, is of great interest for the

verification of such protocols. We are also currently implementing these protocols in Maude.

Acknowledgments: We thank Elie Bursztein, Iliano Cervesato, Patrick Lincoln, Joshua Guttman, Catherine Meadows, Dale Miller, John Mitchell, Paul Rowe, and Carolyn Talcott for helpful discussions. This material is based upon work supported by the MURI program under AFOSR Grant No. FA9550-08-1-0352 and upon work supported by the MURI program under AFOSR Grant No. FA9550-11-1-0137. Additional support for Scedrov from NSF Grant CNS-0830949 and from ONR grant N00014-11-1-0555. Nigam was partially supported by the Alexander von Humboldt Foundation and CNPq. Kanovich was partially supported by the EPSRC.

References

1. R. M. Amadio, D. Lugiez, and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theor. Comput. Sci.*, 290(1):695–740, 2003.
2. S. Brands and D. Chaum. Distance-bounding protocols. In EUROCRYPT 1994.
3. I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell and A. Scedrov. A Meta-Notation for Protocol Analysis. In *CSFW*, 1999, p.55-69.
4. A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*. Springer, 2007.
6. S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, 1971.
7. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
8. N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.
9. J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.
10. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On protection in operating systems. In *SOSP*, 1975.
11. M. Kanovich, T. B. Kirigin, V. Nigam, and A. Scedrov. Bounded memory Dolev-Yao adversaries in collaborative systems. *Inf. Comput.* Accepted for Publication. An extended abstract appeared in FAST 2010.
12. M. Kanovich, T. B. Kirigin, V. Nigam, and A. Scedrov. Progressing collaborative systems. In *FCS-PrivMod*, 2010.
13. M. Kanovich, T. B. Kirigin, V. Nigam, and A. Scedrov. Bounded Memory Protocols and Progressing Collaborative Systems (Technical Report). <http://www.nigam.info/docs/fcs13-tr.pdf>.
14. M. Kanovich, P. Rowe, and A. Scedrov. Policy compliance in collaborative systems. In *CSF*, 2009.
15. M. I. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, C. L. Talcott, and R. Perovic. A rewriting framework for activities subject to regulations. In *RTA*, 2012.
16. M. I. Kanovich, P. Rowe, and A. Scedrov. Collaborative planning with confidentiality. *J. Autom. Reasoning*, 46(3-4):389–421, 2011.
17. C. Meadows, R. Poovendran, D. Pavlovic, L. Chang, and P. F. Syverson. Distance bounding protocols: Authentication logic analysis and collusion attacks. In *Advances in Information Security*, 2007.
18. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.
19. V. Nigam, T. B. Kirigin, A. Scedrov, C. L. Talcott, M. I. Kanovich, and R. Perovic. Towards an automated assistant for clinical investigations. In *IHI*, 2012.
20. M. Rusinowitch and M. Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theor. Comput. Sci.*, 299(1-3):451–475, 2003.

A Multi-Role Translation of Protocol Narration into the Spi-Calculus with Correspondence Assertions^{*}

Eijiro Sumii and Yuji Sato

Graduate School of Information Sciences, Tohoku University
sumii@ecei.tohoku.ac.jp

Abstract. We present an interpretation of protocol narrations by means of translation into the spi-calculus. Our translation allows participants to play multiple roles in parallel, leading to a more general interpretation that considers a wider range of attacks than previous work. We test the validity of our translation by introducing correspondence assertions [Woo and Lam, S&P 1993] to both the protocol narrations and the spi-calculus, and verifying a number of examples by using SpiCA2 [Dahl, Kobayashi, Sun, and Hüttel, ATVA 2011], a sound and automatic type-based verifier of correspondence assertions.

1 Introduction

Security protocols are often written in the so-called *narration* notation (e.g. [4, 9]). For instance, a “repaired” version of the Wide Mouthed Frog protocol [2, Section 3.2.4] can be written like:

1. $A \rightarrow S : A$
2. $S \rightarrow A : N_S$
3. $A \rightarrow S : A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}$
4. $S \rightarrow B : ()$
5. $B \rightarrow S : N_B$
6. $S \rightarrow B : \{S, A, B, K_{AB}, N_B\}_{K_{BS}}$
7. $A \rightarrow B : A, \{M\}_{K_{AB}}$

(In this paper, we write $()$ for the “dummy” message. It was written $*$ in [2].) If literally read, the narration above says

1. Principal A sends message A to principal S .
2. Principal S sends message N_S to principal A .
3. Principal A sends message $A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ to principal S .
4. ...

and so forth. However, this reading is rather superficial and describes only a small part of the actual behavior of each principal. For example:

^{*} Draft as of June 9, 2013

- In step 1, the “server” S should take the name A as a *parameter* to the rest of its actions.
- In step 2, S should *freshly generate* the “nonce” N_S .
- In step 3, S should *decrypt* the ciphertext $\{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ and “check” the last element N_S (as well as the three elements A , A , and B).

To bridge this gap, translations into variants of (a subset of) the spi-calculus [2], based on the *knowledge* of each principal at each point of the protocol, have been proposed [3, 11]. The basic ideas of the translations are as follows:

- When a principal X receives a message M that X does not yet know (i.e., M is not in the knowledge of X at the point of the protocol), X learns M (i.e., M is added to the knowledge of X for the rest of the protocol).
- When a principal X receives a message M that X already knows, X checks whether M is equal to what X knows (if not, X stops).
- When a principal X sends a message M that X does *not* know, X freshly generates M and adds M to its knowledge.¹
- When a principal X receives a ciphertext $\{M\}_{K^+}$ of which X knows the decryption key K^- (same as K^+ in the case of symmetric encryption), X decrypts the ciphertext and behaves as if it received the plaintext M .

For example, if the initial knowledge of A , B , and S is $\{A, B, S, K_{AS}, M\}$, $\{A, B, S, K_{AS}, K_{BS}\}$, and $\{A, B, S, K_{BS}\}$, respectively, and if K_{AS} and K_{BS} are secret (or “the initial knowledge of the attacker” is $\{A, B, S, M\}$), the narration above can be translated into the spi-calculus process

$$\nu K_{AS}. \nu K_{BS}. (A \mid B \mid S)$$

where

$$\begin{aligned}
A &= \mathbf{net!}A \mid & (i) \\
&\mathbf{net?}N_S. & (ii) \\
&\nu K_{AB}. \mathbf{net!}(A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}) \mid & (iii) \\
&\mathbf{net!}(A, \{M\}_{K_{AB}}) \mid & (iv) \\
&0 \\
\\
B &= \mathbf{net?}(). & (i) \\
&\nu N_B. \mathbf{net!}N_B \mid & (ii) \\
&\mathbf{net?}c_1. & (iii) \\
&\mathbf{decrypt} \ c_1 \ \mathbf{is} \ \{S_1, A_1, B_1, K_{AB}, N_{B1}\}_{K_{BS}} \ \mathbf{in} & (iv) \\
&\mathbf{check} \ (S_1, A_1, B_1, N_{B1}) \ \mathbf{is} \ (S, A, B, N_B) \ \mathbf{in} & (v) \\
&\mathbf{net?}(A_2, c_2). & (vi) \\
&\mathbf{check} \ A_2 \ \mathbf{is} \ A \ \mathbf{in} & (vii) \\
&\mathbf{decrypt} \ c_2 \ \mathbf{is} \ \{M\}_{K_{AB}} \ \mathbf{in} & (viii) \\
&0
\end{aligned}$$

¹ In [3], freshly generated messages are explicitly declared “for the sake of clarity”(p. 487).

$S = \text{net?}A_3.$	(i)
$\text{check } A_3 \text{ is } A \text{ in}$	(ii)
$\nu N_S. \text{net!}N_S \mid$	(iii)
$\text{net?}(A_4, c_3).$	(iv)
$\text{check } A_4 \text{ is } A \text{ in}$	(v)
$\text{decrypt } c_3 \text{ is } \{A_5, A_6, B_1, K_{AB}, N_{S1}\}_{K_{AS}} \text{ in}$	(vi)
$\text{check } (A_5, A_6, B_1, N_{S1}) \text{ is } (A, A, B, N_S) \text{ in}$	(vii)
$\text{net!}() \mid$	(viii)
$\text{net?}N_B.$	(ix)
$\text{net!}\{S, A, B, K_{AB}, N_B\}_{K_{BS}} \mid$	(x)
0	

(in this paper, we use ! and ? for output and input, respectively; for the sake of brevity, we also use pattern matching notations on tuples). The key points of the translation are as follows:

- In line (ii) of A and line (ix) of S , the received nonces N_S and N_B are respectively added to the knowledge of the receivers.
- In line (iv) and (viii) of B and line (vi) of S , the received ciphertexts c_1 , c_2 , and c_3 are decrypted with the known keys K_{BS} , K_{AB} , and K_{AS} , respectively.
- In line (v) and (vii) of B and line (ii), (v), and (vii) of S , the integrity of all the known messages are checked when received (or decrypted).
- In line (iii) of A , line (ii) of B , and line (iii) of S , the key K_{AB} and the nonces N_B and N_S , respectively, are freshly generated.

However, this interpretation still suffers from the following limitations:

- The principals B and S assumes a particular A and refuses to talk with other principals. This is especially problematic for the “server” S , which usually should process requests from multiple clients.
- Each principal plays only a single, fixed role (for once). Even if we replicate the translated processes A , B , and S , they still cannot play any other role.

To see a consequence of these limitations, consider the following (broken) variant of the protocol:

1. $A \rightarrow S : A$
2. $S \rightarrow A : N_S$
3. $A \rightarrow S : A, \{B, K_{AB}, N_S\}_{K_{AS}}$
4. $S \rightarrow B : ()$
5. $B \rightarrow S : N_B$
6. $S \rightarrow B : \{A, K_{AB}, N_B\}_{K_{BS}}$
7. $A \rightarrow B : A, \{M\}_{K_{AB}}$

Note that the ciphertexts in step 3 and 6 are “simplified” from $\{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ and $\{S, A, B, K_{AB}, N_B\}_{K_{BS}}$ to $\{B, K_{AB}, N_S\}_{K_{AS}}$ and $\{A, K_{AB}, N_B\}_{K_{BS}}$, respectively. As a result, the protocol becomes insecure when run *in parallel* with the following session of the *same protocol* in the *other direction* (i.e., the roles

Protocols π	$::= \alpha_1; \dots; \alpha_n$
Actions α	$::= X \rightarrow Y : M \mid X \text{ begins } M \mid X \text{ ends } M$
Messages M	$::= v \mid (M, N) \mid \mathbf{inl}(M) \mid \mathbf{inr}(M) \mid M^+ \mid M^- \mid \{M\}_N$
Variables v	$::= X_{Y_1 \dots Y_n}$
Atoms X	$::= A, B, S, x, y, K, N, \dots$

Fig. 1. Syntax of protocol narrations with correspondence assertions

of A and B are swapped).

- 1'. $B \rightarrow S : B$
- 2'. $S \rightarrow B : N'_S$
- 3'. $B \rightarrow S : B, \{A, K_{BA}, N'_S\}_{K_{BS}}$
- 4'. $S \rightarrow A : ()$
- 5'. $A \rightarrow S : N_A$
- 6'. $S \rightarrow A : \{B, K_{BA}, N_A\}_{K_{AS}}$
- 7'. $B \rightarrow A : B, \{M'\}_{K_{BA}}$

Specifically, the attacker can substitute the nonce N_B in step 5 with N'_S in 2', and the ciphertext $\{A, K_{AB}, N_B\}_{K_{BS}}$ in step 6 with $\{A, K_{BA}, N'_S\}_{K_{BS}}$ in 3', tricking B into using K_{BA} instead of K_{AB} in step 7. This flaw could be fixed by introducing “type tags” into the ciphertexts $\{B, K_{AB}, N_S\}_{K_{AS}}$ and $\{A, K_{AB}, N_B\}_{K_{BS}}$ of the protocol—like $\{\mathbf{inl}(B, K_{AB}, N_S)\}_{K_{AS}}$ and $\{\mathbf{inr}(A, K_{AB}, N_B)\}_{K_{BS}}$ —but the problem here is that the previous translation cannot reflect this attack because of the one-to-one assignment of roles to principals.

In this paper, we propose an improved translation of protocol narrations into (a subset of) the spi-calculus, where *every* principal can play *every* role, getting rid of such limitations as above. We furthermore extend our translation to allow insider attacks (i.e., some of the principals may be malicious). We test the validity of our translations by type-checking the translated processes with SpiCA2 [5], a sound and automatic type-based verifier of correspondence assertions [7, 8, 12] in spi-calculus.

The rest of this paper is structured as follows. Section 2 gives the syntax of our protocol narrations and spi-calculus, both extended with correspondence assertions. Section 3 defines the translation, Section 4 gives an example, and Section 5 shows experimental results. Section 6 extends the translation with malicious participants and Section 7 concludes with discussions.

2 Syntax

The syntax of our protocol narrations—extended with correspondence assertions to be used by SpiCA2 after translation—is given in Figure 1. A protocol π is a sequence $\alpha_1; \dots; \alpha_n$ of actions. An action α is either a transmission $X \rightarrow Y : M$ of message M from principal X to Y , or one of the correspondence assertions $X \text{ begins } M$ and $X \text{ ends } M$. A message M is either a variable v , a pair (M, N)

Processes $P ::=$	0	$ $	$\nu v.P$	$ $	$M!N$	$ $	$M?v.P$	$ $	$P Q$	$ $	$*P$
			$ $	$\text{check } M \text{ is } N \text{ in } P$	$ $	$\text{decrypt } M \text{ is } \{v\}_N \text{ in } P$					
			$ $	$\text{case } M \text{ is } \text{inl}(v).P \parallel \text{inr}(w).Q$	$ $	$\text{split } M \text{ is } (v,w) \text{ in } P$					
			$ $	$\text{match } M \text{ is } (N,v) \text{ in } P$	$ $	$\text{begin } M.P$	$ $	$\text{end } M$			

Fig. 2. Syntax of spi-calculus with correspondence assertions

of messages, a tagged message $\text{inl}(M)$ or $\text{inr}(M)$, one of the key pairs M^+ and M^- , or a ciphertext $\{M\}_N$. We assume that a ciphertext encrypted with v , v^+ , and v^- can respectively be decrypted only with v , v^- , and v^+ (as in the standard Dolev-Yao model [6]). We often make v^+ public while keeping v^- private, and sometimes use encryption with v^- for signing (and decryption with v^+ for verification). A variable v has the form $X_{Y_1 \dots Y_n}$ for some $n \geq 0$, where X, Y_1, \dots, Y_n are a kind of “subvariables” called atoms. This will be useful for translating a parametrized variable (e.g., K_{AS} was parametrized over A in the protocols above) into a dynamic look-up.

The syntax of spi-calculus with correspondence assertions (input for SpiCA2 [5]) is given in Figure 2. Process 0 does nothing. $\nu v.P$ generates a fresh name, binds v to it, and executes P . $M!N$ sends message N to channel M , while $M?v.P$ receives a message from channel M , binds v to it, and executes P . $P|Q$ runs P and Q in parallel, and $*P$ spawns an infinite number of parallel P . $\text{check } M \text{ is } N \text{ in } P$ compares M and N , and executes P if they are equal (or stops if not). $\text{decrypt } M \text{ is } \{v\}_N \text{ in } P$ decrypts the ciphertext M with N , binds v to the decrypted plaintext and executes P (or stops if the decryption fails). case and split processes destructs tagged and paired messages, respectively. $\text{match } M \text{ is } (N,v) \text{ in } P$ compares N and the first element of the pair M , and if they are equal, binds v to the second element, and executes P (or stops if not). Although match can be implemented by using split and check , it is given a special typing rule in SpiCA2. Finally, $\text{begin } M$ and $\text{end } M$ are correspondence assertions. (The operational semantics of processes is straightforward [5] and omitted in this paper.)

3 The Translation

In this section, we present our translation of narrations in a “top-down” order according to the syntax in Figure 1.

3.1 Translation of protocols

Given the *initial knowledge* I of participants, which is a partial mapping to messages from names A, B, S, \dots of participants in the narration, a protocol

$\pi = \alpha_1; \dots; \alpha_n$ is translated to the spi-calculus process $\mathcal{T}(\pi)$ as follows:

$$\begin{aligned} \mathcal{T}(\pi) = & * \nu p. \texttt{*part}!p \mid \\ & \nu \texttt{db}. \nu \texttt{dbplus}. \nu \texttt{dbminus}. \\ & (\texttt{*part}?p_1. \texttt{part}?p_2. \nu K_{p_1 p_2}. \texttt{*db}!((p_1, p_2), K_{p_1 p_2}) \mid \\ & \texttt{*part}?p. \nu K_p. (\texttt{*dbplus}!(p, K_p^+) \mid \texttt{*dbminus}!(p, K_p^-) \mid \texttt{*net}!K_p^+) \mid \\ & \prod_{X \in \text{dom}(I)} \mathcal{T}_X(\pi)) \end{aligned}$$

The first line $*\nu p. \texttt{*part}!p$ generates an infinite number of names of participants and keeps sending them to the channel \texttt{part} . As emphasized in the introduction, our translation assigns multiple roles to each participant; thus, after the translation, the number of participants p_1, p_2, \dots (which is infinite!) does not match the number of roles A, B, S, \dots .

The second line $\nu \texttt{db}. \nu \texttt{dbplus}. \nu \texttt{dbminus}$ creates three secret channels \texttt{db} , \texttt{dbplus} , and $\texttt{dbminus}$ for an ideal “key database,” represented by the third and fourth lines. The third line then keeps receiving two names of participants ($\texttt{*part}?p_1. \texttt{part}?p_2$), freshly generates a symmetric key ($\nu K_{p_1 p_2}$), and keeps sending it to \texttt{db} with the two participant names ($\texttt{*db}!((p_1, p_2), K_{p_1 p_2})$). This process is somewhat different from a realistic key database in that it generates an *infinite* number of $K_{p_1 p_2}$ (instead of just one) even for the *same* p_1 and p_2 . This discrepancy is okay as far as sound (but incomplete) verification of safety properties (such as no failure of correspondence assertions) is concerned, since *more* behavior is allowed, not less.

Similarly, the fourth line keeps receiving a participant name ($\texttt{*part}?p$), generates a fresh name (νK_p), and keeps sending the asymmetric key pair to \texttt{dbplus} and $\texttt{dbminus}$ with the participant name ($\texttt{*dbplus}!(p, K_p^+) \mid \texttt{*dbminus}!(p, K_p^-)$) as well as sending the public key to an open network ($\texttt{*net}!K_p^+$). Again, it is fine for our purpose that the process generates an infinite number of key pairs for each principal.

The last line spawns the translations $\mathcal{T}_A(\pi), \mathcal{T}_B(\pi), \mathcal{T}_S(\pi), \dots$ (defined below) of each role A, B, S, \dots (drawn from the domain of the initial knowledge I) in parallel.

3.2 Translation of roles

A role X in protocol $\pi = \alpha_1; \dots; \alpha_n$ is translated as

$$\begin{aligned} \mathcal{T}_X(\alpha_1; \dots; \alpha_n) = & \texttt{*part}?p_1. \dots \texttt{part}?p_m. \\ & \mathcal{T}_X(\rho_1, \alpha_1)(\lambda \rho_2. \\ & \mathcal{T}_X(\rho_2, \alpha_2)(\lambda \rho_3. \\ & \dots \\ & \mathcal{T}_X(\rho_{n-1}, \alpha_{n-1})(\lambda \rho_n. \\ & \mathcal{T}_X(\rho_n, \alpha_n)(\lambda \rho_{n+1}. \\ & 0)) \dots)) \\ & \text{where } \{Y_1, \dots, Y_m\} = \{Y \mid Y \in I(X)\} \\ & \text{and } \rho_1 = \{Y_1 \mapsto p_1, \dots, Y_m \mapsto p_m\} \end{aligned}$$

where $\mathcal{T}_X(\rho_i, \alpha_i)$ is the translation of action α_i for role X with *knowledge* ρ_i , which is a partial mapping from messages in the narration to messages in the translated process. The translated process first receives the names p_1, \dots, p_m of principals of role Y_1, \dots, Y_m (drawn from the initial knowledge $I(X)$ of principals of role X), where the knowledge ρ_1 maps Y_1, \dots, Y_m to p_1, \dots, p_m in the rest of the translation. Since the knowledge may increase by each action, the translation $\mathcal{T}_X(\rho_i, \alpha_i)$ of action α_i in fact takes a continuation $\lambda\rho_{i+1} \dots$ and applies it to the increased knowledge. (We adopt continuation passing style to simplify the definitions.)

3.3 Translation of actions

Action $Y \rightarrow Z : M$, Y *begins* M , and Y *ends* M of role X are translated by case analysis on whether Y or Z matches X . On one hand, if $Y = X$, the translated process $\mathcal{S}_X(\rho, M)(\lambda\sigma.(\dots\sigma^*(M)\dots c[\sigma]))$ looks up the key database and freshly generate names to compose the message $\sigma^*(M)$ to send, begin, or end (see Section 3.5). On the other hand, if $Z = X$, the process $\mathbf{net}?x. \mathcal{R}_X(\rho, x, M)c$ checks the received message if it is known, or else adds it to the knowledge (see Section 3.7). In the other cases, the process does nothing, so the continuation c is just applied to the knowledge ρ without change. (We use square brackets $[]$ for continuation application.)

$$\begin{aligned} \mathcal{T}_X(\rho, X \rightarrow Y : M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. (\mathbf{net}!\sigma^*(M) \mid c[\sigma])) && \text{if } Y \neq X \\ \mathcal{T}_X(\rho, Y \rightarrow X : M)c &= \mathbf{net}?x. \mathcal{R}_X(\rho, x, M)c && \text{if } Y \neq X \text{ and } x \text{ fresh} \\ \mathcal{T}_X(\rho, Y \rightarrow Z : M)c &= c[\rho] && \text{if } Y \neq X \text{ and } Z \neq X \\ \mathcal{T}_X(\rho, X \text{ begins } M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. \mathbf{begin } \sigma^*(M). c[\sigma]) \\ \mathcal{T}_X(\rho, Y \text{ begins } M)c &= c[\rho] && \text{if } Y \neq X \\ \mathcal{T}_X(\rho, X \text{ ends } M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. (\mathbf{end } \sigma^*(M) \mid c[\sigma])) \\ \mathcal{T}_X(\rho, Y \text{ ends } M)c &= c[\rho] && \text{if } Y \neq X \end{aligned}$$

3.4 Message composition

The application $\rho^*(M)$ of knowledge ρ to message M is defined just along the structure of M .

$$\begin{aligned} \rho^*(M) &= \rho(M) && \text{if } M \in \text{dom}(\rho) \\ \rho^*((M_1, M_2)) &= (\rho^*(M_1), \rho^*(M_2)) && \text{otherwise} \\ \rho^*(\mathbf{inl}(M)) &= \mathbf{inl}(\rho^*(M)) \\ \rho^*(\mathbf{inr}(M)) &= \mathbf{inr}(\rho^*(M)) \\ \rho^*(M^+) &= (\rho^*(M))^+ \\ \rho^*(M^-) &= (\rho^*(M))^- \\ \rho^*(\{M\}_N) &= \{\rho^*(M)\}_{\rho^*(N)} \end{aligned}$$

3.5 Fresh name generation

The fresh name generation, required for output (and **begin**) of an unknown message, is defined below. In the first line, it tries to compose the message only by looking up the key database (see Section 3.6). If this look-up fails, the translation works along the structure of the composed message, as in line 2 to 7. In the last line, a fresh name w is generated for the unknown variable v , and the knowledge ρ is extended with the new mapping $v \mapsto w$.

$$\begin{aligned}
\mathcal{S}_X(\rho, M)c &= \text{lookup}_X(\rho, M)c \quad \text{if } \text{lookup}_X(\rho, M) \text{ is defined} \\
\mathcal{S}_X(\rho, (M_1, M_2))c &= \mathcal{S}_X(\rho, M_1)(\lambda\sigma. \mathcal{S}_X(\sigma, M_2)c) \quad \text{otherwise} \\
\mathcal{S}_X(\rho, \text{inl}(M))c &= \mathcal{S}_X(\rho, M)c \\
\mathcal{S}_X(\rho, \text{inr}(M))c &= \mathcal{S}_X(\rho, M)c \\
\mathcal{S}_X(\rho, M^+)c &= \mathcal{S}_X(\rho, M)c \\
\mathcal{S}_X(\rho, M^-)c &= \mathcal{S}_X(\rho, M)c \\
\mathcal{S}_X(\rho, \{M\}_N)c &= \mathcal{S}_X(\rho, N)(\lambda\sigma. \mathcal{S}_X(\sigma, M)c) \\
\mathcal{S}_X(\rho, v)c &= \nu w. c[\rho, v \mapsto w] \quad w \text{ fresh}
\end{aligned}$$

3.6 Key database look-up

When a parameterized variable $X_{Y_1 \dots Y_n}$ in the initial knowledge $I(X)$ of principals of role X is needed, the translated process looks it up in the key database as follows. Again, the translation works along the structure of the message M to be composed, as in line 2 to 7 below. In the first line, if M is already composable (i.e., in the knowledge ρ), no look-up is necessary. Otherwise, the key k received from the database is extracted by using **match**, as in the last 6 lines.

$$\begin{aligned}
\text{lookup}_X(\rho, M)c &= c[\rho] \quad \text{if } M \in \text{dom}(\rho) \\
\text{lookup}_X(\rho, (M_1, M_2))c &= \text{lookup}_X(\rho, M_1)(\lambda\sigma. \text{lookup}_X(\sigma, M_2)c) \quad \text{otherwise} \\
\text{lookup}_X(\rho, \text{inl}(M))c &= \text{lookup}_X(\rho, M)c \\
\text{lookup}_X(\rho, \text{inr}(M))c &= \text{lookup}_X(\rho, M)c \\
\text{lookup}_X(\rho, M^+)c &= \text{lookup}_X(\rho, M)c \\
\text{lookup}_X(\rho, M^-)c &= \text{lookup}_X(\rho, M)c \\
\text{lookup}_X(\rho, \{M\}_N)c &= \text{lookup}_X(\rho, N)(\lambda\sigma. \text{lookup}_X(\sigma, M)c) \\
\text{lookup}_X(\rho, K_{YZ})c &= \text{db?}x. \text{match } x \text{ is } ((\rho(Y), \rho(Z)), k) \text{ in } c[\rho, K_{YZ} \mapsto k] \\
&\quad \text{if } K_{YZ} \in I(X) \text{ and } x, k \text{ fresh} \\
\text{lookup}_X(\rho, K_Z^+)c &= \text{dbplus?}x. \text{match } x \text{ is } (\rho(Z), k) \text{ in } c[\rho, K_Z^+ \mapsto k] \\
&\quad \text{if } K_Z^+ \in I(X) \text{ and } x, k \text{ fresh} \\
\text{lookup}_X(\rho, K_Z^-)c &= \text{dbminus?}x. \text{match } x \text{ is } (\rho(Z), k) \text{ in } c[\rho, K_Z^- \mapsto k] \\
&\quad \text{if } K_Z^- \in I(X) \text{ and } x, k \text{ fresh}
\end{aligned}$$

3.7 Equality checking and knowledge extension

When a message M is received, and if M can also be composed from the knowledge after key database look-ups, their equality with each other is checked (the

first clause below). Otherwise, pairs and tagged messages—as well as ciphertexts with known keys—are destructed or decrypted, and the contents are checked (the second to fifth clauses, where \hat{N} is defined as $\hat{N}^+ = N^-$, $\hat{N}^- = N^+$, and $\hat{N} = N$ otherwise). Once the message cannot be checked or destructed any further, it is added to the knowledge of the receiver (the last clause).

$$\begin{aligned}
\mathcal{R}_X(\rho, x, M)c &= \text{lookup}_X(\rho, M)(\lambda\sigma. \\
&\quad \text{check } \sigma^*(M) \text{ is } x \text{ in } c[\sigma]) \\
&\quad \text{if } \text{lookup}_X(\rho, M) \text{ is defined} \\
\mathcal{R}_X(\rho, x, (M_1, M_2))c &= \text{split } x \text{ is } (y_1, y_2) \text{ in} \quad \text{otherwise} \\
&\quad \mathcal{R}_X(\rho, y_1, M_1)(\lambda\sigma. \\
&\quad \mathcal{R}_X(\sigma, y_2, M_2)c) \quad y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \text{inl}(M))c &= \text{case } x \text{ is} \\
&\quad \text{inl}(y_1). \mathcal{R}_X(\rho, y_1, M)c \parallel \\
&\quad \text{inr}(y_2). 0 \quad y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \text{inr}(M))c &= \text{case } x \text{ is} \\
&\quad \text{inl}(y_1). 0 \parallel \\
&\quad \text{inr}(y_2). \mathcal{R}_X(\rho, y_2, M)c \quad y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\rho, x, \{M\}_N)c &= \text{lookup}_X(\rho, \hat{N})(\lambda\sigma. \\
&\quad \text{decrypt } x \text{ is } \{y\}_{\sigma^*(\hat{N})} \text{ in} \\
&\quad \mathcal{R}_X(\sigma, y, M)c) \\
&\quad \text{if } \text{lookup}_X(\rho, \hat{N}) \text{ is defined and } y \text{ fresh} \\
\mathcal{R}_X(\rho, x, M)c &= c[\rho, M \mapsto x] \quad \text{otherwise}
\end{aligned}$$

It is straightforward to add more checks into the translation above, for instance, when a decryption key K^- is received *after* a ciphertext $\{M\}_{K^+}$ or the corresponding encryption key K^+ . We omitted such “extra” checks in favor of simplicity of the definition, as they were not necessary for our examples.

4 Example

For the sake of presentation, we use n -ary tuples for $n = 0, 3, \dots$ (in addition to pairs) and pattern matching on them. Let us assume the initial knowledge:

$$\begin{aligned}
I(A) &= \{A, B, S, K_{AS}\} \\
I(B) &= \{B, S, K_{BS}\} \\
I(S) &= \{S, K_{AS}, K_{BS}\}
\end{aligned}$$

Note that B does not a priori know A . Note also that S does not know A or B , even though it knows K_{AS} and K_{BS} ! This is fine because K_{AS} and K_{BS} will be looked up from the key database by using the names of A and B received at runtime.

Then, the following (broken) version of the Wide Mouthed Frog protocol

1. $A \rightarrow S : A$
2. $S \rightarrow A : N_S$
3. $A \text{ begins } (A, B, K_{AB})$
4. $A \rightarrow S : \{B, K_{AB}, N_S\}_{K_{AS}} \quad a$
5. $S \rightarrow B : ()$
6. $B \rightarrow S : N_B$
7. $S \rightarrow B : \{A, K_{AB}, N_B\}_{K_{BS}}$
8. $B \text{ ends } (A, B, K_{AB})$

is translated into

```
*νp. *part!p |
νdb.
  (*part?p1. part?p2. νKp1p2. *db!((p1, p2), Kp1p2) |
  A | B | S)
```

(for brevity, the database for asymmetric keys is omitted here), where

```
A = *part?A. part?B. part?S. (*)
  net!A |
  net?NS.
  νKAB. begin (A, B, KAB).
  db?x1. match x1 is ((A, S), KAS) in (***)
  net!{B, KAB, NS}KAS |
  0

B = *part?B. part?S. (*)
  net?().
  νNB. net!NB |
  net?c1.
  db?x2. match x2 is ((B, S), KBS) in (***)
  decrypt c1 is {A, KAB, N'B}KBS in (**)
  check N'B is NB in
  end (A, B, KAB) |
  0

S = *part?S. (*)
  net?A. (**)
  νNS. net!NS |
  net?c2.
  db?x3. match x3 is ((A, S), KAS) in (***)
  decrypt c2 is {B, KAB, N'S}KAS in (**)
  check N'S is NS in
  net!() |
  net?NB.
  db?x4. match x4 is ((B, S), KBS) in (***)
  net!{A, KAB, NB}KBS |
  0
```

The following are highlights of this translation:

- (*) On one hand, the process A is parametrized by the names A , B , and S ; similarly, process B is parameterized by names B and S , and process S by name S .
- (**) On the other hand, process B learns name A *during* the run of the protocol; similarly, S learns A and B at runtime.
- (***) Accordingly, the symmetric key K_{AS} (resp. K_{BS}) shared between A and S (resp. B and S) is looked up from the database at runtime.

5 Experiments

We tested the validity of our translation by verifying its results with SpiCA2 [5], a sound and automatic type-based verifier of correspondence assertions. From the WWW site of SpiCA2 (<http://www.kb.is.s.u-tokyo.ac.jp/~koba/spica2/>), we took 5 protocols using symmetric encryption and 12 using asymmetric.

The results are given in the Table 1 (at the end of the paper, for the sake of page breaks). The columns “expected” and “actual” show the expected and actual results, respectively. “Safe” means that type checking succeeded (i.e., the correspondence assertions would never fail), while “unsafe” means that it failed.

All the actual results match expected ones except for the two “not simply-typed.” They are due to the fact that our translation uses the same public key K^+ for both encryption and signature verification (and the same secret key K^- for both decryption and signing), which does not fit (the “simple” part of) the present type system of SpiCA2. It should be straightforward to adapt the latter to the former (or vice versa).

6 Extension with malicious participants

It is well known that some protocols such as (asymmetric-key version of) Needham-Schroeder [10] are vulnerable to an insider attack, i.e., unsafe when one of the principals is malicious. However, our translation above does not allow such attacks because the channels `db`, `dbplus`, and `dbminus` for the key database are private, i.e., the attacker cannot share any keys with the (good) principals, meaning that it cannot participate in the protocol at all.

To get rid of this limitation, we extend the translation with “bad” participants as follows. First, we separate the name set of bad participants from that of good ones, writing \mathcal{N}^{bad} for the former and \mathcal{N}^{good} for the latter. In the actual translation to SpiCA2, this separation is implemented just by adding an `inl` (for *bad*) or `inr` (for *good*) tag to each name.

The translation of a protocol π then becomes (the changes are underlined>):

$$\begin{aligned} \mathcal{T}(\pi) = & \frac{* \nu p \in \mathcal{N}^{good}. \text{*part!}p \mid * \nu p \in \mathcal{N}^{bad}. \text{*part!}p \mid}{\nu db. \nu dbplus. \nu dbminus.} \\ & (\text{*part?}p_1. \text{part?}p_2. \nu K_{p_1 p_2}. (\text{*db!}((p_1, p_2), K_{p_1 p_2}) \mid \\ & \quad \text{if } p_1 \in \mathcal{N}^{bad} \vee p_2 \in \mathcal{N}^{bad} \text{ then } \text{*net!}K_{p_1 p_2}) \mid \\ & \frac{\text{*part?}p. \nu K_p. (\text{*dbplus!}(p, K_p^+) \mid \text{*dbminus!}(p, K_p^-) \mid \text{*net!}K_p^+ \mid \\ & \quad \text{if } p \in \mathcal{N}^{bad} \text{ then } \text{*net!}K_p^-) \mid}{\prod_{X \in dom(I)} \mathcal{T}_X(\pi)} \end{aligned}$$

The first line generates two kinds of participant names rather than just one. The fourth and sixth lines publish “private” keys if they belong to bad participants so that the attacker can use them.

Then, the translation of a protocol $\pi = \alpha_1; \dots; \alpha_n$ for principals of role X is

$$\begin{aligned} \mathcal{T}_X(\alpha_1; \dots; \alpha_n) = & \text{*part?}p_1. \dots \text{part?}p_m. \\ & \mathcal{T}_X(\underline{b}_1, \rho_1, \alpha_1)(\lambda(\underline{b}_2, \rho_2). \\ & \mathcal{T}_X(\underline{b}_2, \rho_2, \alpha_2)(\lambda(\underline{b}_3, \rho_3). \\ & \dots \\ & \mathcal{T}_X(\underline{b}_{n-1}, \rho_{n-1}, \alpha_{n-1})(\lambda(\underline{b}_n, \rho_n. \\ & \mathcal{T}_X(\underline{b}_n, \rho_n, \alpha_n)(\lambda(\underline{b}_{n+1}, \rho_{n+1}). \\ & 0)) \dots)) \\ & \text{where } \underline{b}_1 = (\{p_1, \dots, p_m\} \subseteq \mathcal{N}^{good}) \\ & \text{and } \{Y_1, \dots, Y_m\} = \{Y \mid Y \in I(X)\} \\ & \text{and } \rho_1 = \{Y_1 \mapsto p_1, \dots, Y_m \mapsto p_m\} \end{aligned}$$

where the translation of each action α_i passes around a Boolean value b_i that represents whether *all* participants involved in the current session is good. This is necessary because, if any of the participants is bad, we will never execute any **end** assertion in this session since there is no hope that the bad participant executes the corresponding **begin** assertion. The rest of the changes are thus (the other definitions remain unchanged):

$$\begin{aligned} \mathcal{T}_X(\underline{b}, \rho, X \rightarrow Y : M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. (\text{net!}\sigma^*(M) \mid c[(\underline{b}, \sigma)])) \quad \text{if } Y \neq X \\ \mathcal{T}_X(\underline{b}, \rho, Y \rightarrow X : M)c &= \text{net?}x. \mathcal{R}_X(\underline{b}, \rho, x, M)c \quad \text{if } Y \neq X \text{ and } x \text{ fresh} \\ \mathcal{T}_X(\underline{b}, \rho, Y \rightarrow Z : M)c &= c[(\underline{b}, \rho)] \quad \text{if } Y \neq X \text{ and } Z \neq X \\ \mathcal{T}_X(\underline{b}, \rho, X \text{ begins } M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. \text{begin } \sigma^*(M). c[(\underline{b}, \sigma)]) \\ \mathcal{T}_X(\underline{b}, \rho, Y \text{ begins } M)c &= c[(\underline{b}, \rho)] \quad \text{if } Y \neq X \\ \mathcal{T}_X(\underline{true}, \rho, X \text{ ends } M)c &= \mathcal{S}_X(\rho, M)(\lambda\sigma. (\text{end } \sigma^*(M) \mid c[(\underline{b}, \sigma)])) \\ \mathcal{T}_X(\underline{b}, \rho, Y \text{ ends } M)c &= c[(\underline{b}, \rho)] \quad \text{if } \underline{b} = \underline{false} \text{ or } Y \neq X \end{aligned}$$

$$\begin{aligned}
\mathcal{R}_X(\underline{b}, \rho, x, M)c &= \text{lookup}_X(\rho, M)(\lambda\sigma. \\
&\quad \text{check } \sigma^*(M) \text{ is } x \text{ in } c[(\underline{b}, \sigma)]) \\
&\quad \text{if } \text{lookup}_X(\rho, M) \text{ is defined} \\
\mathcal{R}_X(\underline{b}, \rho, x, (M_1, M_2))c &= \text{split } x \text{ is } (y_1, y_2) \text{ in} && \text{otherwise} \\
&\quad \mathcal{R}_X(\underline{b}, \rho, y_1, M_1)(\lambda(\underline{b}', \sigma)). \\
&\quad \mathcal{R}_X(\underline{b}', \sigma, y_2, M_2)c && y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \text{inl}(M))c &= \text{case } x \text{ is} \\
&\quad \text{inl}(y_1). \mathcal{R}_X(\underline{b}, \rho, y_1, M)c \parallel \\
&\quad \text{inr}(y_2). 0 && y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \text{inr}(M))c &= \text{case } x \text{ is} \\
&\quad \text{inl}(y_1). 0 \parallel \\
&\quad \text{inr}(y_2). \mathcal{R}_X(\underline{b}, \rho, y_2, M)c && y_1, y_2 \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \{M\}_N)c &= \text{lookup}_X(\rho, \hat{N})(\lambda\sigma. \\
&\quad \text{decrypt } x \text{ is } \{y\}_{\sigma^*(\hat{N})} \text{ in} \\
&\quad \mathcal{R}_X(\underline{b}, \sigma, y, M)c) \\
&\quad \text{if } \text{lookup}_X(\rho, \hat{N}) \text{ is defined and } y \text{ fresh} \\
\mathcal{R}_X(\underline{b}, \rho, x, \underline{A})c &= c[((x \in \mathcal{N}^{\text{good}}) \wedge \underline{b}, (\rho, A \mapsto x))] && \text{if } A \in \text{dom}(I) \\
\mathcal{R}_X(\underline{b}, \rho, x, M)c &= c[(\underline{b}, (\rho, M \mapsto x))] && \text{otherwise}
\end{aligned}$$

It requires some trick to make SpiCA2 accept this translation: as mentioned above, the distinction of “bad participant names \mathcal{N}^{bad} and “good” ones $\mathcal{N}^{\text{good}}$ can be implemented by tagging, but then it often becomes the case that the type of an element of a tuple depends on the *tag* of another element of the same tuple; for instance, in the ciphertext $\{S, A, B, K_{AB}, N_B\}_{K_{BS}}$ of message 6 of the first protocol in Section 1, K_{AB} may be private or public, depending on whether A is good or bad, i.e., tagged by `inl` or `inr`. Such dependency is beyond the power of standard dependent type system as in SpiCA2. To address this problem, we move all `inl` and `inr` tags to the outside of tuples as far as possible (e.g., rewriting $\{\text{inl}(A), K_{AB}\}_{K_{BS}}$ to $\{\text{inl}(A, K_{AB})\}_{K_{BS}}$) and “normalize” (strange) dependent sums like $\Sigma x : \mathcal{N}^{\text{bad}} + \mathcal{N}^{\text{good}}. \text{if } x \in \mathcal{N}^{\text{bad}} \text{ then public else private}$ to simple sums like $(\mathcal{N}^{\text{bad}} \times \text{public}) + (\mathcal{N}^{\text{good}} \times \text{private})$, roughly speaking.

With this trick above, the results in Table 1 in Section 5 remain unchanged even under the presence of malicious participants. This is somewhat surprising because the extended translation allows more attacks. We conjecture that this is only a coincidence of the particular examples of protocols and the type system of SpiCA2, but further investigation is due.

7 Conclusions

We developed an interpretation of protocol narrations as a translation into the spi-calculus, and tested its validity by means of correspondence assertions and their verification.

From the translation, it is obvious that the full power of spi-calculus is not used. One may therefore argue that the target language of the translation can be

simplified. While this is true, we believe that our translation into the spi-calculus (with correspondence assertions) is already simple enough. Moreover, the full power of spi-calculus would be useful for the *attacker* and the *environment* of a protocol.

Another natural question is whether our translation is “correct.” Since there is no standard formal semantics of protocol narrations,² and since our translation is a *definition* of the meaning of protocol narrations, trying to *prove* its correctness seems pointless. However, a more direct semantics of protocol narrations is indeed desirable.

Security properties other than correspondence assertions (authenticity)—such as secrecy [1]—should also be considered in future work.

References

1. Abadi, M.: Secrecy by typing in security protocols. *Journal of the ACM* 46(5), 749–786 (1999), preliminary version appeared in *Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 611–638, 1997
2. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. *Information and Computation* 148(1), 1–70 (1999), preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997
3. Briaïs, S., Nestmann, U.: A formal semantics for protocol narrations. *Theoretical Computer Science* 389(3), 484–511 (2007), preliminary version appeared in *Trustworthy Global Computing, Lecture Notes in Computer Science*, Springer-Verlag, vol. 3705, pp. 163–181, 2005
4. Clark, J., Jacob, J.: A survey of authentication protocol literature: Version 1.0. <http://www-users.cs.york.ac.uk/~jac/papers/drareview.ps.gz> (1997)
5. Dahl, M., Kobayashi, N., Sun, Y., Hüttel, H.: Type-based automated verification of authenticity in asymmetric cryptographic protocols. In: *Proceedings of the 9th international conference on Automated technology for verification and analysis*. *Lecture Notes in Computer Science*, vol. 6996, pp. 75–89. Springer-Verlag (2011)
6. Dolev, D., Yao, A.C.: On the security of public key protocols. *IEEE Transactions on Information Theory* 29(2), 198–208 (1983)
7. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. *Journal of Computer Security* (2003), to appear. Extended abstract appeared in *14th IEEE Computer Security Foundations Workshop*, pp. 145–159, 2001.
8. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security* (2004), to appear. Extended abstract appeared in *15th IEEE Computer Security Foundations Workshop*, pp. 77–91, 2002.

² Briaïs and Nestmann [3] gave a formal semantics by translation into “executable narrations” which “closely correspond to terms in a quite restricted fragment of the spi-calculus” (p. 500). Independently, Sumii et al. [11] defined a (very) similar translation into an (again similar) subset of the spi-calculus, as outlined in Section 1. The former did not consider key databases (nor, in the first place, principals parametrized by the names of other principals, like the server *S* in Section 1), and neither of them considered multiple roles for a single principal.

9. Menezes, A.J., van Oorshot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)
10. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. Communications of the ACM 21(12), 993–999 (1978)
11. Sumii, E., Tatsuzawa, H., Yonezawa, A.: Translating security protocols from informal notation into spi calculus. IPSJ Transactions on Programming 45(SIG 12(PRO 23)), 1–10 (2004), abstract and figures in English, main text in Japanese
12. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: IEEE Symposium on Security and Privacy. pp. 178–194 (1993)

Table 1. Results of experiments with SpiCA2

Protocol	Expected	Actual
A simple handshake using a symmetric key	safe	safe
Woo and Lam’s authentication protocol using a symmetric key	safe	safe
Otway and Ree’s key exchange protocol using a symmetric key	safe	safe
Flawed wide mouth frog protocol	unsafe	unsafe
Fixed variant of wide mouth frog protocol	safe	safe
POSH (public out, secret home) protocol using an asymmetric key	safe	safe
SOPH (secret out, public home) protocol using an asymmetric key	safe	safe
SOSH (secret out, secret home) protocol using an asymmetric key	safe	safe
A three-party protocol that cannot be typed in Gordon and Jeffrey’s type system	safe	not simply-typed
Cremers and Mauw’s generalized Needham-Schroeder-Lowe protocol	safe	safe
ISO Public Key Two-Pass Unilateral Authentication Protocol	safe	safe
Needham-Schroeder protocol (flawed, hence untypable)	unsafe	unsafe
Needham-Schroeder-Lowe protocol (Lowe’s fix, 3-message version)	safe	safe
Needham-Schroeder-Lowe protocol (Lowe’s fix, 7-message version)	safe	not simply-typed
NSL protocol (optimized version)	safe	safe
NSL protocol (with secret)	safe	safe
NSL protocol (with secret and optimization)	safe	safe

Author Index

A			McIver, Annabelle	17
Alvim, Mario S.	2		Meinicke, Larissa	17
Arapinis, Myrto	50		Morgan, Carroll	17
B			N	
Ban Kirigin, Tajana	52		Ngo, Minh	18
			Nigam, Vivek	52
C			R	
Chen, Chen	49		Rocchetto, Marco	51
D			Ryan, Mark	50
Dalle Vedove, Giacomo	51		S	
E			Sato, Yuji	68
Espinoza, Barbara	17		Scedrov, Andre	2, 52
G			Schneider, Fred B.	2
Gadyatskaya, Olga	18		Smith, Geoffrey	17
Gampe, Andreas	34		Smyth, Ben	50
			Sumii, Eihiro	68
J			V	
Jia, Limin	49		Viganò, Luca	51
K			Volpe, Marco	51
Kanovich, Max	52		Von Ronne, Jeffery	34
Köpf, Boris	1		X	
L			Xu, Hao	49
Loo, Boon	49		Z	
Luo, Cheng	49		Zhou, Wenchao	49
M				
Massacci, Fabio	18			