Proceedings

Foundations of Computer Security

Affiliated with LICS'03

Ottawa, Canada
June 26–27, 2003

Edited by
Iliano Cervesato

Contents

Low-Level Primitives

Invited Talk

Language-Based Security

Preface

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

This workshop continues a tradition, initiated with the Workshops on Formal Methods and Security Protocols — FMSP — in 1998 and 1999 and then the Workshop on Formal Methods and Computer Security — FMCS — in 2000, and now the Foundations of Computer Security Workshop — FCS – in 2002, of bringing together formal methods and the security community. The aim of this particular workshop is to provide a forum for continued activity in this area, to bring computer security researchers in contact with the LICS community, and to give LICS attendees an opportunity to talk to experts in computer security.

FCS received 18 submissions. The review phase selected 9 of them for presentation.

Many people have been involved in the organization of the workshop. The Program Committee did an outstanding job at selecting the papers to be presented, in particular given the short review time. Amy Felty and Phil Scott, our interface to LICS, turned a potential bureaucratic nightmare into a smooth ride. Finally we are grateful to the authors, the invited speakers and the attendees who make this workshop an enjoyable and fruitful event.

Iliano Cervesato

FCS'03 Program Chair

Workshop Committees

Program Committee

David Basin, ETH Zürich, Switzerland
Iliano Cervesato (chair), ITT Industries, USA
Hubert Comon, ENS Cachan, France
Edward Felten, Princeton University, USA
Andrew Gordon, Microsoft Research, UK
Jonathan Herzog, MITRE Corporation, USA
Fabio Massacci, University of Trento, Italy
Catherine Meadows, Naval Research Laboratory, USA
John Mitchell, Stanford University, USA
Andrei Sabelfeld, Cornell University, USA
Pierangela Samarati, University of Milano, Italy
Andre Scedrov, University of Pennsylvania, USA
Vitaly Shmatikov, SRI International, USA

Part I

Logical Foundations

Encryption as an abstract data-type: An extended abstract

Dale Miller

INRIA/Futurs & École polytechnique, France

dale.miller@inria.fr

Abstract

At the Dolev-Yao level of abstraction, security protocols can be specified using multisets rewriting. Such rewriting can be modeled naturally using proof search in linear logic. The linear logic setting also provides a simple mechanism for generating nonces and session and encryption keys via eigenvariables. We illustrate several additional aspects of this direct encoding of protocols into logic. In particular, encrypted data can be seen naturally as an abstract data-type. Entailments between security protocols as linear logic theories can be surprisingly strong. We also illustrate how the well-known connection in linear logic between bipolar formulas and general formulas can be used to show that the asynchronous model of communication given by multiset rewriting rules can be understood, more naturally as asynchronous process calculus (also represented directly as linear logic formulas). The familiar proof theoretic notion of interpolants can also serve to characterize communication between a role and its environment.

1 Introduction

When the topic of specifying and reasoning about security protocols attracts the attention of programming language researchers, it is common to find them turning to process calculi, such as CSP or the spi-calculus [AG99, Low96], automata, and even typed λ-calculus [SP01]. Proof search (that is, the foundations of logic programming), however, has a number of properties that are attractive when one attempts to specify and analyze security protocols. We list a few of these advantages here and devote the rest of this paper to develop these benefits further.

Formal analysis of security protocols is largely based on a set of assumptions commonly referred to as the Dolev-Yao model [DY83], an abstraction that supports symbolic execution and reasoning. It has been observed in various places (for example, [DMT98, CDL$^+$99]) that this abstract can be realized well using multiset rewriting. Given that it is well-known that proof search in linear logic provides a declarative framework for specifying multiset rewriting [GG90, HM94, CDL$^+$99], a rather transparent start at representing security primitives is available with proof search in linear logic.

Proof theory comes with a primitive, declarative, and well understood notion of "newness" and "freshness" via the technical devices of *eigenvariable* (newness in proofs) [Gen69] and λ-abstraction [Chu41] (newness in terms). It is appealing to try to use these devices rather than the more *ad hoc* approach to freshness, say, in the BAN logic [BAN89]. Proof search has been used successfully to specify transitions in the π-calculus [MPW92]: *ad hoc* provisos needed to make certain that names and scope were maintained correctly can be encoded in a meta-logic by mapping names to term-level abstractions (λ-abstractions) and to proof-level abstractions (eigenvariables) [Mil03].

Proof search in linear logic has been studied extensively and proof systems exist that give strong normal forms to how search can actually proceed [And92, Mil96]. As a result, significant information is known about the structure of cut-free proofs: since these are the "traces" of protocols and attackers, this structural information is a great aid in reasoning about possible computation paths. The normal forms also provide blueprints for building high-level interpreters for linear logic specifications. Furthermore, the foundations of proof search is a current topic of active research [Gir00] and we can expect that advances there will find applications to this particular domain of computer science.

The proof search paradigm comes equipped with notions of abstract data-types [Mil89] and higher-order predicate abstractions [NM90], all features that we shall draw upon in this paper. Given that these abstraction mechanism all result from aspects of logic, there is no problem in understanding the interaction between abstractions and other aspects of the logic (for example, with multiset rewriting). The problems surrounding "feature interaction" are not present in well designed logics.

Finally, logic comes with a built-in notions of entailment and of equivalence and it is interesting to see if these notions can be useful in reasoning about computation. This notion of logical equivalence would, for example, replace the notion of structural equivalence in process calculus [Mil92]. When one allows rich forms of higher-order quantification, logical entailment is able to relate surprising different specifications.

2 Multiset rewriting in proof search

To model multiset rewriting we shall use a subset of linear logic called *process clauses* in [Mil92]. The closed formula $\forall\bar{x}[G \multimap H]$ is a *process clause* (*clause* for short) if G and H are formulas composed of $\bot$, $⅋$, and $\forall$, and all free variables of G are free in H. Here, $⅋$ encodes the multiset constructor and $\bot$ encodes the empty multiset, H is the *head* of the clause, and G is the *body* of the clause. We will follow the common practice from the logic programming literature of writing the head of a clause first by reversing the sense of the implication: that is, the clause above is written as $\forall\bar{x}[H \circ\!\!- G]$. Kanovich [Kan92, Kan94] introduced a similar set of formulas he calls *linear Horn clauses*: these are essentially process clauses in contrapositive form (replace $⅋$, $\forall$, $\bot$, and $\circ\!\!-$ with $\otimes$, $\exists$, $\mathbf{1}$, and $\multimap$, respectively). We avoid using Kanovich's term here since it is contrary to the usual use of the term *Horn clause* in the logic programming literature where eigenvariables are not part of proof search.

Using simple linear logic equivalences, process clauses can be written in normal form as

$$\forall x_1 \ldots \forall x_i[a_1 ⅋ \cdots ⅋ a_m \circ\!\!- \forall y_1 \ldots \forall y_j[b_1 ⅋ \cdots ⅋ b_n]]$$

where $i, j, n, m \geq 0$ and $a_1, \ldots, a_m, b_1, \cdots, b_n$ are atoms. It is in this form that we generally present process clauses.

In this abstract, we choose sequents of the form $\Sigma : \Psi \longrightarrow \Gamma$ where Σ is the signature (declaration of eigenvariables) and Ψ and Γ are multisets of formulas (linear/bounded maintenance). To illustrate multiset rewriting using the right-hand context, consider the rewriting rule

$$a, b \Rightarrow c, d, e,$$

which specifies that a multiset should be rewritten by removing one occurrence each of a and b and then have one occurrence each of c, d, and e added.

To rewrite the right-hand multiset using the rule above, simply backchain over the clause $c ⅋ d ⅋ e \multimap a ⅋ b$: the full details of such a backchaining looks like the following:

$$\dfrac{\dfrac{\dfrac{\Sigma : \Psi \longrightarrow c, d, e, \Gamma}{\Sigma : \Psi \longrightarrow c, d ⅋ e, \Gamma}}{\Sigma : \Psi \longrightarrow c ⅋ d ⅋ e, \Gamma} \quad \dfrac{\dfrac{}{\Sigma; \cdot \xrightarrow{a} a} \quad \dfrac{}{\Sigma; \cdot \xrightarrow{b} b}}{\Sigma; \cdot \xrightarrow{a ⅋ b} a, b}}{\dfrac{\Sigma; \Psi \xrightarrow{c ⅋ d ⅋ e \multimap a ⅋ b} a, b, \Gamma}{\Sigma : \Psi, c ⅋ d ⅋ e \multimap a ⅋ b \longrightarrow a, b, \Gamma}}$$

Left-introduction rules are applied to the formula on the sequent arrow (instead of general formulas in the left-hand context). The sub-proofs on the right are responsible for deleting from the right-hand context one occurrence each of the atoms a and b while the subproof on the left is responsible for inserting one occurrence each of the atoms c, d, and e and continuing the computation. Thus, the proof fragment above will be simply written as

$$\frac{\Sigma : \Psi \longrightarrow c, d, e, \Gamma}{\Sigma : \Psi, c ⅋ d ⅋ e \multimap a ⅋ b \longrightarrow a, b, \Gamma.}$$

We interpret this fragment of a proof as a rewriting of the multiset a, b, Γ to the multiset c, d, e, Γ.

Backchaining over the clause

$$\forall x_1 \ldots \forall x_i[a_1 ⅋ \cdots ⅋ a_m \circ\!\!- \forall y_1 \ldots \forall y_j[b_1 ⅋ \cdots ⅋ b_n]]$$

is equivalent to the following simple inference rule

$$\frac{\Sigma, y_1, \ldots, y_j : \Psi \longrightarrow \theta b_1, \ldots, \theta b_n, \Gamma}{\Sigma : \Psi \longrightarrow \theta a_1, \ldots \theta a_m, \Gamma}$$

Here, θ is a substitution mapping the variables $x_1, \ldots, x_n$ to Σ-terms and the variables $y_1, \ldots, y_m$ are not declared in Σ (otherwise we can use α-conversion on the clause before backchaining).

Example 2.1 *Consider the problem of Alice wishing to communicate a value to Bob. The clause*

$$\forall x[(a\ x) \parr b \multimapinv a' \parr (b'\ x)]$$

illustrates how one might synchronize Alice's role $(a\ x)$ *with Bob's role* b. *In one, atomic step, the synchronization occurs and the value* x *is transfered from Alice, resulting in her continuation* a', *to Bob, resulting in his continuation* $(b'\ x)$. *If a server is also involved, one can imagine the clause being written as*

$$\forall x[(a\ x) \parr b \parr s \multimapinv a' \parr (b'\ x) \parr s],$$

assuming that the server's state is unchanged through this interaction. As in most examples in this abstract, we shall assume the familiar Prolog convention of writing top-level implications in specification clauses in their reverse direction.

As this example illustrates, synchronization between roles is easy to specify and can trivialize both the nature of communication and the need for security protocols entirely. (For example, if such secure communications is done atomically, there is no need for a server s in the above clause.) While the clause in this example might *specify* a desired communication, it cannot be understood as capturing more low-level and realistic aspects of communications. In distributed settings, synchronization actually only takes place between roles and networks. Our main use of multiset rewriting will involve more restricted clauses with weaker assumptions about communications: these will involve synchronizations between roles and network messages (a model for asynchronous communications) and not between roles and other roles (a model of synchronous communications). Clauses such as those displayed above, however, can serve as *specifications* of what a given protocol might be shown to satisfy.

3 Encoding security protocols

The following encoding of security protocols follows to a large extent that given in [CDL$^+$99, CDL$^+$00] for the system MSR.

The type i is used to encode messages. Primitive objects, such as integers, strings, and nonces, are all constructors of i. The tupling operator $\langle\cdot,\cdot\rangle$, for pairing data together, has type $i \to i \to i$. Expressions such as $\langle\cdot,\cdot,\ldots,\cdot\rangle$ denote pairing associated to the right. One additional constructor for i is presented in Section 4.

A *network message* is encoded as an atomic formula of the form $\mathsf{N}(t)$, where $\mathsf{N}(\cdot)$ is a predicate of type $i \to o$ and t (of type i) is the actual data of the message. (Following [Chu40], we use o to denote the type of formulas.) As we shall see, the state of the network will be a multiset of such atomic formulas.

The roles (e.g., Alice and Bob) and the formulas that describe them are rather complicated in MSR: we provide some definitions inspired by the definitions given in [CDL$^+$99, CDL$^+$00]. A *role identifier* is a symbol, say, ρ. For some number $n \geq 1$ and for $i = 1,\ldots,n$, the pair ρ_i of an identifier and an index is a *role state predicate*. These state predicates are used to encode internal states of a role as a protocol progresses. A *role state atom* is an atomic formula of the form $\rho_i(x_1,\ldots,x_m)$ where $x_1,\ldots,x_m$ are distinct variables and ρ_i is a role state predicate. A *role clause* is a process clause

$$\forall x_1\ldots\forall x_i[a_1 \parr \cdots \parr a_m \multimapinv \forall y_1\ldots\forall y_j[b_1 \parr \cdots \parr b_n]]$$

where $i,j,n,m \geq 0$. Here, the *head* of such as clause is the formula $a_1 \parr \cdots \parr a_m$ and the *body* is $\forall y_1\ldots\forall y_j[b_1 \parr \cdots \parr b_n]$. Role clauses also have the following restrictions: all the atoms $a_1,\ldots,a_m,b_1,\ldots,b_n$ are either network messages or a role state atoms such that (1) there is at most one role state atom in the head and at most one in the body; (2) if there is a role state atom in the head, say, $\rho_i(\bar{t})$ and a role state atom in the body, say, $\rho'_j(\bar{s})$, then ρ and ρ' must be the same role identifier and $i < j$. In other words, a role clause only involves a single role (and possibly network messages) and when moving from the head to the body, the index of the role must increase. As a consequence of the restrictions on role clauses, roles cannot synchronize with other roles directly and one role cannot evolve into another role. Condition (1) allows for process creation (no role state atom in the head) and process deletion (no role state atom in the body). Condition (2) above implies is that all agents have finite runs [CDL$^+$99, CDL$^+$00]. A final restriction on role clauses is that all variables free in the body of the clause must be free in the role state atom in the head of the clause.

A *role theory* is a linear logic formula of the form

$$\exists x_1\ldots\exists x_r\ [C_1 \otimes \cdots \otimes C_s],$$

where $r,s \geq 0$, $C_1,\ldots,C_s$ are role clauses, where $x_1,\ldots,x_r$ are variables of type i or $i \to i$, and whenever C_i and C_j have the same role state predicate in their head then $i = j$. This latter condition implies that agents in protocols

Message 1	$A \longrightarrow S:$	A, B, n_A
Message 2	$S \longrightarrow A:$	$\{n_A, B, k_{AB}, \{k_{AB}, A\}_{k_{BS}}\}_{k_{AS}}$
Message 3	$A \longrightarrow B:$	$\{k_{AB}, A\}_{k_{BS}}$
Message 4	$B \longrightarrow A:$	$\{n_B\}_{k_{AB}}$
Message 5	$A \longrightarrow B:$	$\{n_B, \mathit{Secret}\}_{k_{AB}}$

Figure 1: A conventional presentation of the Needham-Schroeder protocol.

are deterministic. This is a condition that can easily be relaxed within linear logic if non-deterministic agents are of interest. Fortunately, this restriction does not restrict the class of intruders (testers) that we consider briefly in Section 7 since intruders can be grouped together in multisets and interleaving provides non-determinism.

Existential quantification like that surrounding role theories are used in logic programming to provide for abstract data-types and here they will serve as local constants shared by certain role clauses. In particular, shared keys between, say Alice and a trusted server, will be existentially quantified in this way with a variable of type $i \rightarrow i$. The use of existential quantifier at type $i \rightarrow i$ is explained next.

4 Encryption as an abstract data-type

Encryption keys will be encoded using function symbols of type $i \rightarrow i$. Since such keys will need to be given scope, they will be quantified either existentially over role theories or universally in role clauses. Using higher-order quantification over data constructors is the usual way to specify *abstract data-types* within logic programming [Mil89]. Since we will be allowing quantification of higher-order type, proof search will be slightly more complicated than if we restricted ourselves to only first-order quantification. For example, the (meta-level) equations for α, β, and η conversions are assumed, although no other equations on the type i are assumed. (As is customary with typed λ-terms, η is assumed since there seems to be no good reason to distinguish, say, the encryption key k form the expression $(\lambda w.kw)$.) As discussed in [Mil02], higher-order quantification can add greatly to the expressive strength of specification, but when done carefully, it does not need to add to the complexity of proof search. The remaining constructor for the type i is $\cdot^\circ$ of type $(i \rightarrow i) \rightarrow i$: this constructor is used to coerce an encryption key back into a data item, and in this way, a role can place a key into a network message. (We will not introduce application $\mathit{app} : i \rightarrow (i \rightarrow i)$, the dual operation to $\cdot^\circ$, since the expression $(\mathit{app}\, k^\circ\, x)$ will be written simply as $(k\, x)$.) This approach to "encryption as an abstract data-type" is a departure from MSR.

Consider the following specification that contains three occurrences of encryption keys.

$$\exists k_{as} \exists k_{bs} [\begin{array}{lll} a_1\langle M, S\rangle & \circ\!- & a_2 S \parr \mathsf{N}(k_{as}\ M). \\ b_1 T \parr \mathsf{N}(k_{bs}\ M) & \circ\!- & b_2\langle M, T\rangle. \\ s_1() \parr \mathsf{N}(k_{as}\ P) & \circ\!- & \mathsf{N}(k_{bs}\ P). \end{array}]$$

(Here as elsewhere, quantification of capital letter variables is universal with scope limited to the clause in which the variable appears.) In this example, Alice (a_1, a_2) communicates with Bob (b_1, b_2) via a server (s_1). To make the communications secure, Alice uses the key k_{as} while Bob uses the key k_{bs}. The server is deleted immediately after it translates one message encrypted for Alice to a message encrypted for Bob. The use of the existential quantifiers helps establish that the occurrences of keys, say, between Alice and the server and Bob and the server, are the only occurrences of that key. Even if more principals are added to this system, these occurrences are still the only ones for these keys. Thus, the existential quantifier helps in determining the static or lexical scope of key distribution. Of course, as protocols are evaluated (that is, a proof is searched for), keys may extrude their scope and move freely onto the network. This dynamic notion of scope extrusion is similar to that found in the π-calculus [MPW92] and is modeled here similar to an encoding of the π-calculus into linear logic found in [Mil92].

Example 4.1 *The Needham-Schroeder Shared Key protocol presented in Figure 1 [SC01] and the rough translation of it into linear logic given in Figure 2 provides another example. Notice that two shared keys are used in this example and that the server creates a new key that is placed within data and is then used by Alice and Bob to communicate directly. It is a simple matter to show that this protocol implements the specification (taken from Example 2.1):*

$$\forall x[a_1(x) \parr b_1() \parr s_1() \circ\!- a_4() \parr b_3(x)].$$

$\exists k_{as} \exists k_{bs} \{$

$$\begin{array}{rcl}
a_1(S) & \circ\!- & \forall na.\, a_2(na, S) \parr \mathsf{N}(\langle alice, bob, na\rangle). \\
a_2(N, S) \parr \mathsf{N}(k_{as}\langle N, bob, K^\circ, En\rangle) & \circ\!- & a_3(N, K, S) \parr \mathsf{N}(En). \\
a_3(Na, Key, S) \parr \mathsf{N}(Key\ Nb) & \circ\!- & a_4() \parr \mathsf{N}(Key\ \langle Nb, S\rangle). \\
b_1() \parr \mathsf{N}(k_{bs}\ \langle Key^\circ, alice\rangle) & \circ\!- & \forall nb.b_2(nb, Key) \parr \mathsf{N}(Key\ nb). \\
b_2(Nb, Key) \parr \mathsf{N}(Key\langle Nb, S\rangle) & \circ\!- & b_3 S. \\
s_1 \parr \mathsf{N}(\langle alice, bob, N\rangle) & \circ\!- & \forall k.\mathsf{N}(k_{as}\langle N, bob, k^\circ, k_{bs}\langle k^\circ, alice\rangle\rangle).
\end{array}$$

$\}$

Figure 2: Encoding the Needham-Schroeder protocol.

The linear logic proof of this starts with the multiset $a_1(c)$, $b_1()$, $s_1()$ on the right of the sequent arrow (for some "secret" eigenvariable c) and then reduces this back to the multiset $a_4() \parr b_3(c)$ simply by performing a simple "execution" of the logic program in Figure 2. Notice that the $\forall$ used in the bodies of clauses in this protocol are used both for nonce creation (at type i) and encryption key creation (at type $i \to i$).

Example 4.2 *Consider the following two clauses for Alice.*

$$aK^\circ \parr \mathsf{N}(K\ M) \circ\!- a'M. \qquad (3.1)$$
$$a \parr \mathsf{N}(K\ M) \circ\!- a'M. \qquad (3.2)$$

In the first case, she possesses an encryption key and uses it to decrypt a network message. In the second case, it appears that she is decrypting a message without knowing the key, an inappropriate behavior, of course. Technically, this clause is not a proper role clause (since there is a variable M that is not free in the role state predicate in the head a). In any case, it is interesting to consider (3.2) for a moment. Notice that (3.2) is logically equivalent (and, hence, operationally indistinguishable using proof search) to both of the formulas

$$\forall M \forall X[a \parr \mathsf{N}(X) \circ\!- a'M] \quad \textit{and} \quad \forall X[a \parr \mathsf{N}(X) \circ\!- \exists M.a'M].$$

This last clause clearly illustrates that Alice is not actually decoding an existing message but is simply guessing (using $\exists$) at some data value M, and continues with that guess as $a'M$. If one thinks operationally instead of declaratively about proof search involving clause (3.2), we would consider possible unifiers for matching the pattern $(K\ M)$ with a network message, say, $(k\ s)$, for two constants k and s. Unification yields exactly the following three different unifiers:

$$[M \mapsto (k\ s), K \mapsto \lambda w.w] \quad [M \mapsto s, K \mapsto k] \quad [M \mapsto M, K \mapsto \lambda w.(k\ s)]$$

Thus, M can be bound to either $(k\ s)$ or s or any term: in other words, M can be bound to any expression of type i.

By using higher-order quantification, logical entailment is strengthened and can help in reasoning about role clauses and theories.

Example 4.3 *Consider the two clauses*

$$a_1 \circ\!- \forall k.\mathsf{N}(k\ m) \quad \textit{and} \quad a_1 \circ\!- \forall k.\mathsf{N}(k\ m').$$

Each of these clauses specify that Alice can take a step that generates a new encryption key and then outputs a message (either m or m') using that encryption key. Since Alice has no continuation, no one, not even Alice will be able to decode this message. It should be the case that these two clauses are "operationally" similar since they both generate a "junk message." In fact, it is an easy matter to show that these two clauses are logically equivalent. A proof that the first implies the second contains a subproof of the sequent

$$\forall k.\mathsf{N}(k\ m') \longrightarrow \forall k.\mathsf{N}(k\ m),$$

and this is proved by introducing an eigenvariable, say c, on the right and the term $\lambda w.(c\ m)$ on the left.

5 Abstracting internal states

The following example illustrates that using existential quantification over *predicates* (in particular, role state predicates) allows interesting rewriting of the structure of role theories.

Example 5.1 (Reducing n-way to 2-way synchronization) *General n-way synchronization ($n \geq 2$) can be rewritten using 2-way synchronization by the introduction of new, intermediate, and hidden predicates as is allowed in role theories. For example, the following two formulas are logically equivalent.*

$$\exists l_1 \exists l_2. \left\{ \begin{array}{c} a \parr b \multimapinv l_1 \\ l_1 \parr c \multimapinv l_2 \parr e \\ l_2 \multimapinv d \parr f \end{array} \right\} \quad \dashv\vdash \quad a \parr b \parr c \multimapinv d \parr e \parr f$$

The clause on the right specifies a 3-way synchronization and the spawning of 3 new atoms whereas the formula on the left is limited to rewriting at most two atoms into at most 2 atoms. The proof of the forward entailment in linear logic is straightforward while the proof of the reverse entailment involves the two higher-order substitutions of $a \parr b$ for $\exists l_1$ and $d \parr f$ for $\exists l_2$. As long as we are using logical entailment, these two formulas are indistinguishable and can be used interchangeably in all contexts. If instead we could observe possible failures in the search for proofs, then it is possible to distinguish these formulas: consider the search for a proof of a sequent containing a and b but not c. Since linear logic does not observe such failures, this kind of observation cannot be internalized.

Existential quantification over program clauses can also be used to hide predicates encoding roles. In fact, one might argue that the various restrictions on sets of process clauses (no synchronization directly with atoms encoding roles, no role changing into another role, etc) might all be considered a way to enforce locality of predicates. Existential quantification can, however, achieve this same notion of locality, but much more declaratively.

Example 5.2 (Hiding role state predicates) *The following two formulas are logically equivalent:*

$$\exists\, a_2, a_3. \left\{ \begin{array}{l} a_1 \parr \mathsf{N}(m_0) \multimapinv a_2 \parr \mathsf{N}(m_1) \\ a_2 \parr \mathsf{N}(m_2) \multimapinv a_3 \parr \mathsf{N}(m_3) \\ a_3 \parr \mathsf{N}(m_4) \multimapinv a_4 \parr \mathsf{N}(m_5) \end{array} \right\} \quad \dashv\vdash$$

$$a_1 \parr \mathsf{N}(m_0) \multimapinv (\mathsf{N}(m_1) \multimapinv (\mathsf{N}(m_2) \multimapinv (\mathsf{N}(m_3) \multimapinv (\mathsf{N}(m_4) \multimapinv (\mathsf{N}(m_5) \parr a_4)))))$$

The changing of polarity that occurs when moving to the premise of a $\multimapinv$ flips expressions from output (e.g., $\mathsf{N}(m_1)$) to input (e.g., $\mathsf{N}(m_2)$), etc. Thus, by hiding intermediate roles state predicates, it is possible to rewrite a role theory into a different style of formula that seems quite natural.

6 Asynchronous and synchronous connectives

The observation that abstracting over internal states results in an equivalent syntax with nested $\multimapinv$ suggests an alternative syntax for roles. Consider the following syntactic categories of linear logic formulas:

$$H ::= A \mid \perp \mid H \parr H \mid \forall x.H$$

$$K_O ::= H \mid H \multimapinv K_I \mid \forall x.K_O \qquad K_I ::= H \multimapinv K_O \mid \forall x.K_I$$

Here, A denotes the class of atomic formulas encoding network messages and formulas belonging to the class H denote bundles of messages that are used as either input or output to the network. Formulas belonging to the classes K_I and K_O can have deep nesting of implications and that nesting changes phases from input to output and back to input. The reason to split the class of K formulas into input formulas (K_I) and output formulas (K_O) is to parallel the MSR formalism more closely: in MSR, after a agent does an input, it must do an output (even if there is not messages to output). That is, in MSR, every input action has a continuation but not every output action as a continuation.

A formula in the category K_I is called a *role formula*.

The connectives of linear logic can be classified as asynchronous connective ($\parr$, $\forall$, $\&$, etc) and synchronous connective ($\otimes$, $\exists$, $\oplus$, etc). The dual of a connective in one class is a connective in the other. The formulas of MSR are examples of *bipolar*: these are formulas in which no asynchronous connective is in the scope of a synchronous connective. For example, $Q_1 \parr \cdots \parr Q_m \multimapinv P_1 \parr \cdots \parr P_n$ is logically equivalent to $Q_1 \parr \cdots \parr Q_m \parr (P_1^\perp \otimes \cdots \otimes P_n^\perp)$. Obviously, role formulas are, in general, not bipolars.

$$
\begin{array}{ll}
\text{(Out)} & \forall na.\mathsf{N}(\langle alice,\ bob,\ na\rangle) \multimap \\
\text{(In)} & \quad (\forall Kab \forall En.\mathsf{N}(kas\langle na,\ bob,\ Kab^{\circ},\ En\rangle) \multimap \\
\text{(Out)} & \quad\quad (\mathsf{N}(En) \multimap \\
\text{(In)} & \quad\quad\quad (\forall NB.\mathsf{N}(KabNB) \multimap \\
\text{(Out)} & \quad\quad\quad\quad (\mathsf{N}(Kab(NB,\ secret)))))).
\end{array}
$$

$$
\begin{array}{ll}
\text{(Out)} & \bot \multimap \\
\text{(In)} & \quad (\forall Kab.\mathsf{N}(kbs(Kab^{\circ},\ alice)) \multimap \\
\text{(Out)} & \quad\quad (\forall nb.\mathsf{N}(Kab\, nb) \multimap \\
\text{(In)} & \quad\quad\quad (\mathsf{N}(Kab(nb,\ secret)) \multimap \\
\text{(Cont)} & \quad\quad\quad\quad b\, secret))).
\end{array}
$$

$$
\begin{array}{ll}
\text{(Out)} & \bot \multimap \\
\text{(In)} & \quad (\forall N.\mathsf{N}(\langle alice,\ bob,\ N\rangle) \multimap \\
\text{(Out)} & \quad\quad (\forall k.\mathsf{N}(kas\langle N,\ bob,\ k^{\circ},\ kbs(k^{\circ},\ alice)\rangle))).
\end{array}
$$

Figure 3: The roles of Alice, Bob, and a server

Proposition 6.1 *For every role theory in which only the predicate* $\mathsf{N}(\cdot)$ *is free, there is a collection of role formulas to which it is provably equivalent.*

Proof: This proposition is proved by showing how to remove the existentially quantified role state predicate with maximal index by generating the appropriate higher-order substitution (similar to those produced in Example 5.1). When no more quantified role state predicates remain, the resulting theory is the desired collection of role formulas. □

To illustrate an example of this style of syntax, consider first declaring local all role predicates in the Needham-Schroeder Shared Key protocol in Figure 2. This then yields the logically equivalent presentation in Figure 3. There, three formulas are displayed: the first represents the role of Alice, the second Bob, and the final one the server. (All agents in this Figure are written at the same polarity, in this case, in output mode: since Bob and the server essentially start with inputs, these two agents are negated, meaning they first output nothing and then move to input mode.) Andreoli's compilation method [And92] applied to the formula in Figure 3 yields the formulas in Figure 2: the new constants introduced by compilation are the names used to denote role continuation.

The style of specification given in Figure 3 is similar to that of process calculus: in particular, the implication $\multimap$ is syntactically similar to the dot prefix in, say, CCS. Universal quantification can appear in two modes: in output mode it is used to generate new eigenvariables (similar to the π-calculus restriction operator) and in input mode it is used for variable binding (similar to value-passing CCS). The formula $a \multimap (b \multimap (c \multimap (d \multimap k)))$ can denote processes described as

$$\bar{a} \,||\, (b.\, (\bar{c} \,||\, (d.\ \ldots))) \quad \text{or} \quad a.\, (\bar{b} \,||\, (c.\, (\bar{d} \,||\, \ldots)))$$

depending on which polarity it is being used. This formula and it's negation can also be written without linear implications as follows:

$$a \parr (b^{\perp} \otimes (c \parr (d^{\perp} \otimes \ldots))) \text{ resp, } a^{\perp} \otimes (b \parr (c^{\perp} \otimes (d \parr \ldots))).$$

Once a process with a continuation (that is, one that has an implication) has done an output (input), its continuation is an input (output) process. To see this mechanism in the proof search setting, consider a sequent $\Delta \longrightarrow \Gamma$ where Δ is a multset of K_I formulas and Γ are multisets of K_O formulas (here, we elide the signature associated to a sequent). The right-hand side of sequents involve asynchronous behavior (output) and left-hand side of sequents involve synchronous behavior (input). The two rules involving proof search with implications can be written as follows:

$$\frac{\Delta, K \longrightarrow \Gamma, H, \mathcal{A}}{\Delta \longrightarrow H \multimap K, \Gamma, \mathcal{A}} \qquad \frac{H \longrightarrow \mathcal{A}_1 \qquad \Delta \longrightarrow K, \mathcal{A}_2}{\Delta, H \multimap K \longrightarrow \mathcal{A}_1, \mathcal{A}_2}$$

Here, $\mathcal{A}$ denotes a multiset of atoms (i.e., network messages). Notice that we can assume that the left-introduction rule for $\multimap$ is only done when the right-hand side of the concluding sequent contains at most atomic formulas.

If the three formulas in Figure 3 are placed on the right-hand side of a sequent arrow (with no formulas on the left) then the role formula for Alice will output a message and move to the left-side of the sequent arrow (reading inference rules bottom up). Bob and the server output nothing and move to the left-hand side as well. At that point, the server will need to be chosen for a $\multimap$L inference rule, which will cause it to input the message that Alice sent and then move its continuation to the right-hand side. It will then immediately output another message, and so on.

Various equivalences familiar from the study of asynchronous communication are found in linear logic. For example, if one skips a phase, the two phases can be contracted as follows:

$$p \circ\!- (\bot \circ\!- (q \circ\!- k)) \equiv p \parr q \circ\!- k$$

$$p \circ\!- (\bot \circ\!- \forall x(q\, x \circ\!- k\, x)) \equiv \forall x(p \parr q\, x \circ\!- k\, x)).$$

While the nested presentation of roles is in some sense, more complicated syntax than the MSR (bipolar) format, this presentation certainly has its advantages over MSR. For example, there is only one predicate, namely $\mathsf{N}(\cdot)$, involved in writing out security protocols: role identifiers and role state predicates have disappeared. A role can now be seen as simply a formula and a role theory as simply an existentially quantified tensor of roles.

The following two examples illustrate a difference between the abstractions available in logic with those available in the π-calculus and the spi-calculus.

Example 6.2 (Comparison with the π-calculus) *The π-calculus expression*

$$(x)(\bar{x}m \mid x(y).Py)$$

is (weakly) bisimilar to the expression (Pm). This example is used to show that communication over a hidden channel provides no possible means for the environment to interact. A similar expression can be written as the following expression in linear logic:

$$\forall K[Km \parr (\forall x(Px \multimap Kx) \multimap \bot)].$$

Here, we have abstracted the predicate *K: in a sense, we have abstracted the communication medium itself, and as such, the medium is available only for the particular purpose of communicating the message m from one process to another that is willing to do an input. This expression is logically equivalent to (Pm): the proof that (Pm) implies the displayed formula involves a use of equality (easy to add to the underlying logic in a number of ways) and the higher-order substitution $\lambda w.(w = m) \multimap \bot$ for K.*

Example 6.3 (Comparison with the spi-calculus) *In the spi-calculus, a "public" channel can be used for communicating. To ensure that messages are only "understood" by the appropriate parties, messages are encrypted with keys that are given specific scopes. For example, the expression*

$$(k)(\bar{q}(\{m\}_k) \mid q(y).let\ \{x\}_k = y\ in\ Px)$$

describes a process that is willing to output an encrypted message $\{m\}_k$ on a public channel q and to also input such a message and decode it. The key k is given a scope similar to that given in the π-calculus expression. The linear logic expression, call it B,

$$\forall k[\mathsf{N}(km) \parr (\forall x(Px \multimap \mathsf{N}(kx))) \multimap \bot]$$

is most similar to this spi-calculus expression: here, the network $\mathsf{N}(\cdot)$ corresponds to the public channel q. It is not the case, however, that B is logically equivalent to Pm since linear logic can observe that B can output something on the public channel, that is, $\forall y(\top \multimap \mathsf{N}(y)) \vdash B$ whereas it is not necessarily true that $\forall y(\top \multimap \mathsf{N}(y)) \vdash Pm$ is provable.

7 Interpolants and communications

We now illustrate how proof theory techniques can be applied to reasoning about security protocols.

Notice that role formulas, even collections of them, are not intended to be proved. In general, they are intended to evolve to other role formulas. To help analyze such evolution, we introduce *tests* which are essentially role formulas with an additional primitive for terminating proof search. In linear logic, the syntactic class of tests is written as

$$W ::= \top \mid H \mid H \circ\!- W \mid \forall x.W,$$

which is similar to the definition of K_O except that the additive true $\top$ is allowed as well. Following rather standard methods of using context to characterize a formula's meaning (as in, say, Kripke models or phase spaces), let $\|P\|$ denote the set of all multisets Γ of tests such that $\vdash \Gamma, P$ and define *testing equivalence* $P \simeq Q$ to hold when $\|P\| = \|Q\|$. Since $P \vdash Q$ implies $\|P\| \subseteq \|Q\|$, logically equivalent role formulas are testing equivalent. The converse is not the case.

In order to get a better handle on testing equivalence, it is valuable to find ways to simplify the structure of tests. As they stand now, we need to consider arbitrary implementations of roles that act as intruders. While intruders (tests) might do rather complicated internal actions, it is communications between the principals and the intruder via the network that should characterize the "external" meaning of a role.

Since data and encryption keys are encoded as constructors and eigenvariables, it would appear that we can apply the proof theoretic notion of *interpolant* [Cra57] to help monitor communications. Classically, if $A \vdash B$ then an interpolant is a formula R such that $A \vdash R$ and $R \vdash B$ and all the non-logical symbols occurring in R occur in both A *and* B. In our linear logic setting, we can prove the following theorem. (For a related use of interpolants, see [Mil92].)

Proposition 7.1 *Let Γ be a multiset of role formulas (principals) and let Δ be a multiset of tests (intruders) such that $\vdash \Gamma, \Delta$. This proof has an interpolant: that is, there is a formula R such the non-logical constants occurring in R occur in both Γ and in Δ, and is such that $\vdash \Gamma, R$ and $R \vdash \Delta$.*

Proof Outline. The proof is by induction on the structure of a cut-free proof of $\vdash \Gamma, \Delta$. The base case is the $\top$-R rule. In this case, the tester contains $\top$ and we also set the interpolant for this sequent to be $\top$. The other inference rules can be classified as either asynchronous or synchronous. The synchronous rules are $\forall L$ and $\multimap L$ and the other rules are asynchronous. All the asynchronous rules have exactly one premise and the interpolant associated to the premise is also associated to the conclusion. Now consider the two synchronous cases.

First notice that given the classification of formulas in the endsequent $\longrightarrow \Gamma, \Delta$ as being either principal or test, all formulas in all sequents of the (cut-free) proof can be similarly labeled as being subformulas of either a principal or a test.

Assume that the last inference rule of the proof is the following occurrence of $\multimap L$ and assume that R is an interpolant for the right premise.

$$\frac{H \longrightarrow \mathcal{A}_1 \qquad \Delta, \Gamma \longrightarrow K, \mathcal{A}_2}{\Delta, \Gamma, H \circ\!- K \longrightarrow \mathcal{A}_1, \mathcal{A}_2}$$

Here, Γ is a multiset of principal formulas and Δ is a multiset of tests. Let M_P be the $⅋$ of the principal atoms in $\mathcal{A}_1$ and let M_I be the $⅋$ of the test atoms in $\mathcal{A}_1$: if there are no such atoms of either classification, use $\perp$ as the formula. We now have two cases to consider: either the formula $H \circ\!- K$ comes from a principal or it comes from a test. In the first case, the interpolant associated to the bottom sequent is $M_I ⅋ R$. In the second case, the interpolant for the bottom sequent is $(M_P \circ\!- R)^\perp$.

Consider finally the $\forall L$ inference rule and assume that R is an interpolant for the premise.

$$\frac{Kt, \Gamma, \Delta \longrightarrow \mathcal{A}_P, \mathcal{A}_I}{\forall_\gamma x.Kx, \Gamma, \Delta \longrightarrow \mathcal{A}_P, \mathcal{A}_I}$$

Here, $\mathcal{A}_P$ is a multiset of principal atoms and $\mathcal{A}_I$ is a multiset of test atoms. Assume that the formula R is not an interpolant for the lower sequent since it contains occurrences of non-logical constants that do not occurr in both a principal formula and a test formula. In particular, let $c : \gamma$ be such a non-logical constant. Clearly, c has an occurrence in t. We must now distinquich two cases inorder to abstract away the occurrence of c in R as follows.

Case 1: The occurrence of the formula $\forall_\gamma x.Kx$ is a principal formula. In that case, we can conclude that t contains an occurrence of c, that no principal formula of the lower sequent contains c, and that some test forumula in the lower sequent also contains an occurrence of c. The new interpolant is then $\forall x.R[x/c]$ for some variable x of type γ. Proving the sequent $\forall_\gamma x.Kx, \Gamma \longrightarrow \mathcal{A}_P, \forall x.R[x/c]$ can be reduced to proving $Kt, \Gamma \longrightarrow \mathcal{A}_P, R$ by using a $\forall$-R (instantiating with c) and $\forall$-L rule (instantiating with t). Similarly, proving the sequent $\Delta, \forall x.R[x/c] \longrightarrow \mathcal{A}_I$ can be reduced to proving $\Delta, R \longrightarrow \mathcal{A}_I$ $\forall$-L rule (instantiating with c). Both of these two sequents are assumed provable using the inductive assumption.

Case 2: The occurrence of the formula $\forall_\gamma x.Kx$ is a test formula. In that case, we can conclude that t contains an occurrence of c, that no test formula of the lower sequent contains c, and that some principal forumula in the lower sequent also contains an occurrence of c. The new interpolant is then $\exists x.R[x/c]$ for some variable x of type γ. Proving the sequent $\forall_\gamma x.Kx, \Delta, \exists x.R[x/c] \longrightarrow \mathcal{A}_I$ can be reduced to proving $Kt, \Delta, R \longrightarrow \mathcal{A}_I$ by using a $\exists$-L (instantiating with c) and $\forall$-L rule (instantiating with t). Similarly, proving the sequent $\Gamma \longrightarrow \mathcal{A}_P, \exists x.R[x/c]$ can be reduced to proving $\Delta \longrightarrow \mathcal{A}_P, R$ using $\exists$-L rule (instantiating with c). Both of these sequents are assumed provable using the inductive assumption.

Repeat this abstraction for each possible constant c. The resulting abstraction is then an interpolant for the concluding sequent. In this way, one can build an interpolant inductively over the structure of the cut-free proof.

A more careful reading of the structure of interpolants shows that they have the following structure:

$$M ::= \bot \mid A \mid M \parr M$$

$$R^+ ::= \top \mid \forall x.R^+ \mid M \multimapinv R^- \qquad R^- ::= M \multimapinv R^+ \mid \forall x.R^-.$$

Formulas in the R^+ syntactic category are contain the interpolants and are called *traces*. Clearly, traces are in fact simple tests (W-formulas).

Let $\|P\|_t$ denote the set of all traces R such that $\vdash P, R$ and define *trace equivalence*, $P \simeq_t Q$, as $\|P\|_t = \|Q\|_t$. Since traces are tests, $P \simeq Q$ implies $P \simeq_t Q$. The interpolation theorem provides a simple proof of the converse.

Proposition 7.2 *The relations $\simeq$ and $\simeq_t$ coincide.*

Proof: Assume that $P \simeq_t Q$ and that Γ is a set of testers such that $\vdash P, \Gamma$. By Proposition 7.1, we know that there is a trace R such that $\vdash P, R$ and $R \vdash \Gamma$. Since $P \simeq_t Q$, we know that $\vdash Q, R$ and by cut, we know that $\vdash Q, \Gamma$. Thus $P \simeq Q$. □

Thus, the structure of attacks from intruders (testers) can be characterized by using the much simpler notion of (single-threaded) trace. This result is familiar also from process calculus and when interpreted as a theorem of concurrency, it is certainly not new. What is new here is showing that it is a simple consequence of the proof theoretic notion of interpolant.

8 Future Work

This paper shows that the logical foundations of MSR can be enriched and, at the same time, simplified, at least in the sense that some details that were addressed with various non-logical constants (continuation predicates, explicit encryption procedures, etc) can be replaced with logical concepts. As a result, reasoning about the meaning and correctness of protocols should be enhanced. Although some minor examples in the paper illustrate this possibility, much additional work remains to be done to validate this conjecture. I outline here some key topics to develop further.

The Dolev-Yao intruder. The notion of tester introduce in Section 7 is a natural way to provide meaning to a formula (in proof theory) or processes (in concurrency). The exact connection, however, between testing and the Dolev-Yao notion of intruder must be formally established.

Security properties. Generally, the reason to formalize a security protocol is as a first step to establish various kinds of security related properties, such as secrecy and authentication. One starting point for establishing such properties uses reachability analysis, and techniques in [MM02, MMP03], which work directly on logic specifications of the form given here, might be one place to start.

Experimentation. Of course, serious experimentation with protocols and their formal properties should be explored. Given a modern logic programming language such as λProlog that incorporates λ-terms and their unification, eigenvariables, and proof search, it is an easy matter to implement interpreters that can execute and explore simple reachability questions. Designing a deductive system that formally supports reasoning about specifications that contains such computational elements is significantly more difficult.

New quantification. An extension to proof theory described in [Mil03] proposes a new quantifier, $\nabla x.G$, which might be more appropriate for capturing the notion of scoping of new names within a security protocol. It seems likely that ∇ could be used to generate nonces while $\forall$ would continue to be used for encryption keys. The distinction between these quantifiers would not appear in the execution of the operational semantics of protocols but in reasoning about their equivalence and behavior.

Related formalisms. There are obviously many other paradigms to which this work can be related. Given that there is a well known connection between π-calculus style encodings and eigenvariables [Mil92], it would seem quite natural to explore more fully possible connections with the spi-calculus [AG99]. Also, the connection between *strand spaces* [CDKS00, CDL$^+$00, FHG99] and cut-free proofs using role formulas seems likely be strong: one of the relations involved in defining a strand space should be that of subformula within a common role and the other relation should relate two atoms (messages) appearing together in the same initial sequent.

Acknowledgments. I wish to thank Iliano Cervesato, Alwen Tiu, and the anonymous reviewers of previous versions of this paper for their suggestions.

Bibliography

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 99.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Operating Systems Review*, 23(5):1–13, December 1989.

[CDKS00] I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In H. Veith, N. Heintze, and E. Clark, editors, *2000 Workshop on Formal Methods and Computer Security - FMCS'00*, July 2000.

[CDL+99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *12th IEEE Computer Security Foundations Workshop*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[CDL+00] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop*, pages 35–51, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269– 285, 1957.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, June 1998.

[DY83] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[FHG99] F. J. T. Fabrega, C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *J. of Computer Security*, 7(2,3):191–230, 1999.

[Gen69] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[GG90] V. Gehlot and C. Gunter. Normal process representatives. *Fifth IEEE Sym. on Logic in Computer Science*, pages 200–207, Philadelphia, June 1990.

[Gir00] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3), June 2000.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[Kan92] Max Kanovich. Horn programming in linear logic is NP-complete. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 200–210. IEEE Computer Society Press, June 1992.

[Kan94] Max Kanovich. The complexity of Horn fragments of linear logic. *Annals of Pure and Applied Logic*, 69:195–241, 1994.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, eds, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, *LNCS* 1005, pp. 147–166. 1996.

[Mil89] D. Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, June 1989. MIT Press.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *1992 Workshop on Extensions to Logic Programming*, LNCS 660, pages 242–265. Springer-Verlag.

[Mil96] D. Miller. Forum: A multiple-conclusion specification language. *TCS*, 165(1):201–232, September 1996.

[Mil02] D. Miller. Higher-order quantification and proof search. In H. Kirchner and C. Ringeissen, editors, *Proc. of AMAST 2002*, LNCS 2422, pages 60–74, 2002.

[Mil03] D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. To appear in the Proceedings of LICS03. July 2003.

[MM02] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.

[MMP03] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *TCS*, 294(3):411–437, 2003.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992.

[NM90] Gopalan Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[SC01] P. Syverson and I. Cervesato. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.

[SP01] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *14th IEEE Computer Security Foundations Workshop*, June 2001.

A 3-Valued Logic for the Specification and the Verification of Security Properties

Béchir Ktari

LSFM Research Group,
Computer Science Department,
Laval University,
Sainte-Foy, Qc, Canada

bechir.ktari@ift.ulaval.ca

Abstract

The main intent of this paper is the definition of a 3-valued logic for the specification and the verification of security properties. This is a part of a large initiative that we undertook a few years ago to address malicious code detection in commercial off-the-shelf software products. Building on top of the existing approaches, we have established a framework that stems from a combination of self-certified code technology and model-checking techniques. In this paper, we propose a logic with a 3-valued semantics in order to be able to reason under uncertainty. We also endow our logic with a tableau-based proof system that is proven to be finite, and to be sound and complete with respect to the denotational semantics of the logic. Furthermore, we propose some ideas that may lead to important future extensions. For instance, we show how the use of variables and types can lead to a more efficient verification.

1 Motivation

With the advent and the rising popularity of networks, Internet, intranets and distributed systems, security is becoming one of the focal points of research. Basically, the main issue of this research is the detection of malicious code in software, such as logic bombs, Trojan horses, viruses, worms, etc. A preliminary study of the domain [BDD$^+$99] reveals that different approaches could be suitable to address this problem. Among others, certified code [Koz98, MWCG99, Nec97] seems to be the most promising. However, most of the research on certified code have unfortunately focused on simple type safety and memory safety, rather than security issues. We therefore propose a framework that stems from a combination of self-certified code technology and model-checking techniques, in order to extend this approach to the security aspects of a program. The idea is to extract a model from the information contained in the certificate. This model provides an abstraction of the security aspects of the program. Afterwards, this model is matched against a logical specification of security policies.

Nevertheless, in the presence of pointers, alias pointers and function pointers, it is generally difficult if it is not impossible, using static analysis, to find out all function calls within a piece of software. Hence, the model (transition system) associated with this piece of software will certainly contains transitions with unknown actions that may lead to the failure of the verification process.

In this paper, we present our results related to the definition and the use of a logic for the specification and verification of high-level security properties. Our main contribution is the definition of a 3-valued semantics that allows to reason under uncertainty (lack of information). We also endow our logic with a tableau-based proof system that is proven to be finite, sound and complete.

Here is how the rest of this paper is organized. In Section 2, we describe the syntax and the 3-valued denotational semantics of the logic. In Section 3, we present a tableau-based proof system for our logic. Section 4 proposes an improved version of both denotational and tableau-based semantics allowing a more efficient and precise verification. In Section 5, we extend the model in order to enhance the capability of the verification process. Section 6 is devoted to the presentation of related work, while Section 7 discusses some aspects related to the information flow issue as a future research. Finally, a few concluding remarks are ultimately sketched in Section 8.

Table 1: Syntax of the logic.

ψ	::=	tt | ff | $\neg\psi$ | $\psi \vee \psi'$ | $\psi \wedge \psi'$ | $<K>\psi$ | $[K]\psi$ | X | $\mu X.\psi$ | $\nu X.\psi$
K	::=	$\{a_i \mid i = 1..n\}$ | $K - K'$
a_i	::=	*Critical Functions*

2 Logic for security properties

Roughly speaking, a *model-checker* is a procedure that decides whether a given structure M is a model of a logical formula ψ. M corresponds to an abstract model of the program in question, typically represented by a graph structure where the nodes represent the program states and the arcs represent possible transitions between the states. In our work, we use a *labelled transition system* (LTS), where the arcs are annotated with actions.

2.1 Model

The model is defined as a quadruplet $M = \langle \mathcal{S}, \mathcal{A}ct \cup \{\delta\}, \rightarrow, s_0 \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}ct$ is a set of actions ($\delta \notin \mathcal{A}ct$), $\rightarrow \subseteq \mathcal{S} \times (\mathcal{A}ct \cup \{\delta\}) \times \mathcal{S}$ is a transition relation, and $s_0 \in \mathcal{S}$ is the initial state of the model. A transition $(s, a, s') \in \rightarrow$ states that the system can evolve from state s to state s' by performing action a. We adopt the more intuitive notation $s \stackrel{a}{\rightarrow} s'$.

The model also admits δ-transitions which abstract actions that are unknown at compile time. Actually, many information related to control flow can not be discovered statically. For instance, function pointers involve functions that may be unknown at compile time. This is mainly due to the lack of some information, available only at runtime, and to the use of pointers, alias pointers and function pointers.

2.2 Syntax of the logic

In this section we introduce a logic that allows one to specify properties of expressions. The logic we consider may be viewed as a variant of the modal μ-calculus [Koz83], or the Hennessey-Milner Logic with recursion.

The syntax of formulae is presented in Table 1. The logic is refered as L_μ.

The symbols $\neg$, $\vee$ and $\wedge$ respectively represent negation, disjunction, and conjunction. The symbol $<K>$, known as the diamond operator, is a modal operator indexed by the set of actions K. The meaning of modal formulae appeals to the transition behavior of a program. For instance, a program satisfies the formula $<K>\psi$ if it can evolve to some state obeying ψ by performing an action from K. In the same way, the symbol $[K]$ is a modal operator known as the box. A program satisfies the formula $[K]\psi$ if, after the performance of any action in K, the result state satisfies ψ. Variables X range over a set of formula variables $\mathcal{V}$. The formula $\mu X.\psi$ ($\nu X.\psi$) is a recursive formula where the least fixpoint operator μ (greatest fixpoint operator ν) binds all free occurrences of X in ψ. An occurrence of X is free if it is not within the scope of a binding μX or νX. Note that as for the μ-calculus, all occurrences of X in ψ must appear inside the scope of an even number of negations. This is to ensure the existence of fixpoints.

Assuming that $\mathcal{A} \subseteq \mathcal{A}ct$ is a universal set of actions, "-K" abbreviates the set "$\mathcal{A} - K$" and "-" abbreviates the set "-$\emptyset$", which is just the set $\mathcal{A}$. For more convenience, we assume that "all" abbreviates "-". For instance, the formula [-]ff is equivalent to [all]ff. Moreover, within the modalities, we write $a_1, \ldots, a_n$ instead of $\{a_1, \ldots, a_n\}$.

The next section proposes several examples of formulae.

Examples

In this section, we present several examples that are frequently used in the security area. We begin with some simple formulae that express basic behaviors. Note that all formulae are interpreted relatively to an implicit "current state".

- <*createProcess*>true
- [*readPassword*]<*checkPassword*>true
- [*formatDisk,send*]false

The first formula states that action[1] *createProcess* is allowed in the "current state". The second formula states that after every performance of a *readPassword* action, the program can perform at least one *checkPassword* action. The third formula states that no *formatDisk*, nor *send* actions are allowed.

However, all these formulae are not very expressive since they refer to a specific "current state" and not to the entire system. We therefore define some abbreviations that are convenient for the specification of security properties:

- always(ψ) $= \nu X.\psi \wedge$ `[all]` X
- eventually(ψ) $= \mu X.\psi \vee$ `<all>`X
- never(ψ) $= \neg$eventually(ψ)
- loop(a) $= \nu X.$<a>X

Combining abbreviations and fixpoint operators offers much expressiveness:

- never(<*formatDisk,send*>`true`)
- always(`[`*openSocket*`]` eventually(<*closeSocket*>`true`))
- never(loop(*createProcess*))

The first formula states that transitions involving *formatDisk* or *send* are not allowed anywhere in the system. The second formula states that each time *openSocket* is executed, it must be possible to close it in a later state. The third formula states that no loop containing *createProcess* is allowed in the model of the corresponding program.

2.3 Denotational Semantics

Formulae are interpreted over models of the form $M = \langle \mathcal{S}, \mathcal{A}ct \cup \{\delta\}, \rightarrow, s_0 \rangle$, and environments of the form $e = [X_i \mapsto 2^{\mathcal{S}} \times 2^{\mathcal{S}}]$ which map variables X_i to a pair of sets of states. Semantically, formulae of the logic correspond to sets of states for which they are true and to sets of states for which they may be true, including those that are true. This is due to the presence of δ-transitions in the model.

The meaning function $[\![.]\!]_e^M : \mathcal{F} \rightarrow 2^{\mathcal{S}} \times 2^{\mathcal{S}}$ is described[2] in Table 2. The set $\mathcal{S}$ refers to the set of all states in the model, and $\mathcal{F}$ refers to all formulae of L_μ. K_δ refers to $K \cup \{\delta\}$. $\pi_1(p)$ and $\pi_2(p)$ denote respectively the first and second element of a pair p. Therefore, for any formula ψ, the following property holds:

$$[\![\psi]\!]_e^M = (\pi_1([\![\psi]\!]_e^M), \pi_2([\![\psi]\!]_e^M)) \tag{1}$$

Intuitively, all states satisfy or might satisfy the formula `tt` while there are no states that satisfy or might satisfy `ff`. The meaning of a variable X is simply the pair of sets of states that are bound to X in the environment e. Disjunction and conjunction are interpreted in a classical way.

The meaning of formula $\neg\psi$ is a pair of sets of states S_1, S_2 such that the elements of S_1 are states that do not satisfy ψ and those of S_2 are states that might not satisfy ψ. Since the set $\pi_2([\![\psi]\!]_e^M)$ represents states that might satisfy ψ, including those that satisfy the formula, the other states in $\mathcal{S}$ do not satisfy ψ. In the same manner, the set $\pi_1([\![\psi]\!]_e^M)$ represents states that satisfy ψ. Thus, the other states in $\mathcal{S}$, including those of $\pi_2([\![\psi]\!]_e^M) - \pi_1([\![\psi]\!]_e^M)$, do not or might not satisfy ψ.

The meaning of formula $<K>\psi$ is a pair of sets of states s that might evolve, by performing an action a taken from K, to some state s' such that s' is part of the meaning of ψ. The set of states that satisfy this formula is interpreted in a classical way, except for the case where the action set K corresponds to all actions in $\mathcal{A}ct$ (i.e. `all`). In this case, the meaning of the corresponding formula, `<all>`ψ, is a pair of sets of states that can evolve to states satisfying ψ, by performing any action from $\mathcal{A}ct$. Moreover, if a state s evolves to such a state s' by performing a δ abstract action, then it necessarily satisfies the formula.

The set of states that might satisfy $<K>\psi$ is interpreted in a classical way, i.e. it is the set of all states that can evolve, by performing an action taken from K_δ, to some state s' such that $s' \in \pi_2([\![\psi]\!]_e^M)$.

[1] We use abstract actions rather than real APIs (Application Program Interface).

[2] Quantifications have the form (*quantifier bound variable* | *range restriction* : *quantified expression*) (see, *e.g.*, [GS93]); an empty range in a quantification means that the bound variable ranges over all possible values.

Table 2: A 3-valued denotational semantics for the logic.

$$
\begin{array}{ll}
[\![\mathsf{tt}]\!]_e^M & = (\mathcal{S},\mathcal{S}) \\
[\![\mathsf{ff}]\!]_e^M & = (\emptyset,\emptyset) \\
[\![X]\!]_e^M & = e(X) \\
[\![\neg\psi]\!]_e^M & = (\overline{\pi_2([\![\psi]\!]_e^M)}, \overline{\pi_1([\![\psi]\!]_e^M)}) \\
[\![\psi_1 \vee \psi_2]\!]_e^M & = (\pi_1([\![\psi_1]\!]_e^M) \cup \pi_1([\![\psi_2]\!]_e^M), \pi_2([\![\psi_1]\!]_e^M) \cup \pi_2([\![\psi_2]\!]_e^M)) \\
[\![\psi_1 \wedge \psi_2]\!]_e^M & = (\pi_1([\![\psi_1]\!]_e^M) \cap \pi_1([\![\psi_2]\!]_e^M), \pi_2([\![\psi_1]\!]_e^M) \cap \pi_2([\![\psi_2]\!]_e^M)) \\
[\![<K>\psi]\!]_e^M & = (\ \{s \mid (\exists s' \mid s' \in \pi_1([\![\psi]\!]_e^M) : (\exists a \mid a \in K : s \xrightarrow{a} s'))\} \cup \\
& \quad \{s \mid (K = \mathsf{all}) \wedge (\exists s' \mid s' \in \pi_1([\![\psi]\!]_e^M) : s \xrightarrow{\delta} s')\}, \\
& \quad \{s \mid (\exists s' \mid s' \in \pi_2([\![\psi]\!]_e^M) : (\exists a \mid a \in K_\delta : s \xrightarrow{a} s'))\} \\
& \ ) \\
[\![\,[K]\,\psi]\!]_e^M & = (\ \{s \mid (\forall s' \mid : (\forall a \mid a \in K_\delta : s \xrightarrow{a} s' \Rightarrow s' \in \pi_1([\![\psi]\!]_e^M)))\}, \\
& \quad \{s \mid (\forall s' \mid : (\forall a \mid a \in K : s \xrightarrow{a} s' \Rightarrow s' \in \pi_2([\![\psi]\!]_e^M)))\} \cap \\
& \quad \{s \mid (K = \mathsf{all}) \ \Rightarrow\ (\forall s' \mid : s \xrightarrow{\delta} s' \Rightarrow s' \in \pi_2([\![\psi]\!]_e^M))\} \\
& \ ) \\
[\![\mu X.\psi]\!]_e^M & = \bigcap\{(Q,Q') \subseteq (\mathcal{S}\times\mathcal{S}) \mid [\![\psi]\!]_{e[X\mapsto(Q,Q')]}^M \subseteq (Q,Q')\} \\
[\![\nu X.\psi]\!]_e^M & = \bigcup\{(Q,Q') \subseteq (\mathcal{S}\times\mathcal{S}) \mid (Q,Q') \subseteq [\![\psi]\!]_{e[X\mapsto(Q,Q')]}^M\}
\end{array}
$$

Table 3: Simplified syntax of the logic.

$$\psi ::= X \mid \neg\psi \mid <K>\psi \mid \psi\wedge\psi' \mid \nu X.\psi$$

The meaning of formula $[K]\psi$ is a pair of sets of states such that after every action a in K, each result state is part of the meaning of ψ. The set of states that satisfy this formula is interpreted in a classical way. Note that the set of actions that is involved in this definition includes the δ abstract action.

The set of states that might satisfy $[K]\psi$ is also interpreted in a classical way, except that we have to add all those states s that may perform a δ-transition and evolve to states that do not satisfy ψ. This set is added only if $K \neq \mathsf{all}$, since if $K \neq \mathsf{all}$, the state s does not satisfy the formula whatever the instantiated value of δ is.

The meaning of the fixpoint formulae[3] is the same as in the μ-calculus. Hence the greatest fixpoint is given as the union of all post-fixpoints whereas the least fixpoint is the intersection of all pre-fixpoints.

Note that the semantics preserves the classical properties of μ-calculus:

$$\neg\neg\psi \equiv \psi \tag{2}$$

$$\neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2 \tag{3}$$

$$[K]\psi \equiv \neg<K>\neg\psi \tag{4}$$

$$\mu X.\psi \equiv \neg\nu X.\neg\psi[\neg X/X] \tag{5}$$

where $\psi[\psi'/X]$ represents the simultaneous replacement of all free occurrences of X in ψ by ψ'.

Given these properties, the syntax of the logic can be simplified as proposed in Table 3. Note that $\mathsf{ff} \equiv \mu X.X$ and $\mathsf{tt} \equiv \neg\mathsf{ff}$.

3 Tableau-based Proof System

The denotational semantics proposed in Section 2.3 is based on a global model-checking paradigm. In such paradigm, we need to compute all the states that satisfy or might satisfy a given formula ψ in order to determine that such formula

[3] $(Q,Q') \subseteq (R,R') \Leftrightarrow Q \subseteq Q' \wedge R \subseteq R'$

Table 4: Tableau rules for $H \vdash s \star \psi$, where $\star$ in $\{\in, \approx\}$.

$$\text{(R1)}\quad \frac{H \vdash s \star \neg\neg\psi}{H \vdash s \star \psi} \qquad \text{(R2)}\quad \frac{H \vdash s \star \psi_1 \wedge \psi_2}{H \vdash s \star \psi_1 \quad H \vdash s \star \psi_2}$$

$$\text{(R3)}\quad \frac{H \vdash s \star \neg(\psi_1 \wedge \psi_2)}{H \vdash s \star \neg\psi_1} \qquad \text{(R4)}\quad \frac{H \vdash s \star \neg(\psi_1 \wedge \psi_2)}{H \vdash s \star \neg\psi_2}$$

$$\text{(R5)}\quad \frac{H \vdash s \star {<}K{>}\psi}{H \vdash s' \star \psi} \quad \begin{cases} C_1 & \text{if } \star = \in \\ C_1' & \text{if } \star = \approx \end{cases}$$

$$\text{(R6)}\quad \frac{H \vdash s \star \neg{<}K{>}\psi}{H \vdash s_1 \star \neg\psi \quad \ldots \quad H \vdash s_n \star \neg\psi} \quad \begin{cases} C_2 & \text{if } \star = \in \\ C_2' & \text{if } \star = \approx \end{cases}$$

$$\text{(R7)}\quad \frac{H \vdash s \star \nu X.\psi}{H[X \mapsto H(X) \cup \{s\}] \vdash s \star \psi[\nu X.\psi/X]} \quad s \notin H(X)$$

$$\text{(R8)}\quad \frac{H \vdash s \star \neg\nu X.\psi}{H[X \mapsto H(X) \cup \{s\}] \vdash s \star \neg\psi[\nu X.\psi/X]} \quad s \notin H(X)$$

C_1: $s' \in \{s' \mid (\exists a \mid a \in K : s \xrightarrow{a} s')\} \cup \{s' \mid (K = \mathsf{all}) \ \wedge \ s \xrightarrow{\delta} s'\}$
C_2: $\{s_1, \ldots, s_n\} = \{s' \mid (\exists a \mid a \in K_\delta : s \xrightarrow{a} s')\}$
C_1': $s' \in \{s' \mid (\exists a \mid a \in K_\delta : s \xrightarrow{a} s')\}$
C_2': $\{s_1, \ldots, s_n\} = \{s' \mid (\exists a \mid a \in K : s \xrightarrow{a} s')\} \cup \{s' \mid (K = \mathsf{all}) \ \Rightarrow \ s \xrightarrow{\delta} s'\}$

satisfies or might satisfy a model M. However, the local model-checking paradigm does not require that every state in the model be examined. Indeed, only required states are visited. The local model-checking, based on a tableau proof system [SW89], aims to determine that a given state, generally the initial state of a model, satisfies or might satisfy a given formula.

Following [Cle90], we describe, in this section, a tableau-based proof system for establishing when states in a model satisfy or might satisfy formulas of the logic proposed in Table 3. Also, we give three results regarding the finiteness, soundness and completeness of the proposed proof system.

3.1 Tableau System

Table 4 shows the tableau-based proof system. This system is based on the formulae presented in Table 3. The idea is to capture, in a deductive way, whether a given state satisfies or might satisfy a formula ψ. The proof rules operate on sequents[4] of the form $H \vdash s \in \psi$ and $H \vdash s \approx \psi$, where H is a mapping in $[\mathcal{V} \mapsto 2^{\mathcal{S}}]$, s is a state and ψ is a formula. The intended meaning of a sequent $H \vdash s \in \psi$ is that under assumptions H, s satisfies ψ, and the intended meaning of a sequent $H \vdash s \approx \psi$ is that under assumptions H, s might satisfy ψ.

The proofs are conducted in a top-down fashion, i.e. the proof rules are written with conclusions appearing above premises. Moreover, if a sequent σ' results from the application of a rule to a sequent σ, then we say that σ' is a child of σ or that σ is a parent of σ'. We will use the notation $\sigma \rightarrow_R \sigma'$ to denote that σ' is a child of σ using a rule R (Table 4).

A sequent σ has a successful tableau if there exists a finite tableau having σ as a root and its leaves are successful. Following [Cle90], a sequent σ is successful when it meets one of the following conditions:

1. $\sigma = (H \vdash s \in \neg{<}K{>}\psi) \wedge \{s' \mid (\exists a \mid a \in K_\delta : s \xrightarrow{a} s')\} = \emptyset$.
2. $\sigma = (H \vdash s \in \nu X.\psi) \wedge s \in H(X)$.
3. $\sigma = (H \vdash s \approx \neg{<}K{>}\psi) \wedge (\{s' \mid (\exists a \mid a \in K : s \xrightarrow{a} s')\} \cup$
$\{s' \mid (K = \mathsf{all}) \ \Rightarrow \ s \xrightarrow{\delta} s'\} = \emptyset)$.
4. $\sigma = (H \vdash s \approx \nu X.\psi) \wedge s \in H(X)$.

[4] Actually, we use the sequent $H \vdash s \star \psi$, where $\star$ in $\{\in, \approx\}$.

We have established the following three major results whose proofs are reported in [Kta03].

Theorem 3.1 (Finiteness) *For any sequent* $\sigma = (H \vdash s \star \psi)$, $\star$ *in* $\{\in, \approx\}$, *there is a maximum height tableau with root* σ.

Theorem 3.2 (Soundness)

1. $H \vdash s \in \psi$ *has a successful tableau* $\Rightarrow$ $s \in \pi_1(\llbracket\psi\rrbracket_e^{M,(H,[\,])})$.

2. $H \vdash s \approx \psi$ *has a successful tableau* $\Rightarrow$ $s \in \pi_2(\llbracket\psi\rrbracket_e^{M,([\,],H)})$.

Theorem 3.3 (Completeness)

1. $s \in \pi_1(\llbracket\psi\rrbracket_e^{M,(H,[\,])})$ $\Rightarrow$ $H \vdash s \in \psi$ *has a successful tableau.*

2. $s \in \pi_2(\llbracket\psi\rrbracket_e^{M,([\,],H)})$ $\Rightarrow$ $H \vdash s \approx \psi$ *has a successful tableau.*

In order to establish the soundness and the completeness of the proof system, we have introduced a relativized semantics [Cle90, Kta03]. We did so by defining a new semantic function $\llbracket.\rrbracket_e^{M,H} : \mathcal{F} \to 2^{\mathcal{S}} \times 2^{\mathcal{S}}$ for formulae in $\mathcal{F}$, environment e, hypothesis map H and model M. H is a pair of maps, $(H_\in, H_\approx)$, associating sets of states to variables. It turns out that $\llbracket.\rrbracket_e^{M,H} = \llbracket.\rrbracket_e^{M}$ for $H = ([\,],[\,])$.

4 Toward a more Efficient Semantics

With regard to the semantics defined in previous sections, there is a case where this semantics is not as precise as one wants. For instance, let's $M = \langle\{s_0, s_1\}, \{a, b, c\}, s_0 \xrightarrow{\delta} s_1, s_0\rangle$ be a model and $\psi = \texttt{<}a\texttt{>tt} \vee \texttt{<-}a\texttt{>tt}$ be a formula. By definition (Table 2), we have the following:

$$\begin{aligned}\llbracket \texttt{<}a\texttt{>tt} \vee \texttt{<-}a\texttt{>tt}\rrbracket_e^M = {} & (\pi_1(\llbracket \texttt{<}a\texttt{>tt}\rrbracket_e^M) \cup \pi_1(\llbracket \texttt{<-}a\texttt{>tt}\rrbracket_e^M), \\ & \pi_2(\llbracket \texttt{<}a\texttt{>tt}\rrbracket_e^M) \cup \pi_2(\llbracket \texttt{<-}a\texttt{>tt}\rrbracket_e^M))\end{aligned}$$

It is easy to see that: $\llbracket \texttt{<}a\texttt{>tt}\rrbracket_e^M = \llbracket \texttt{<-}a\texttt{>tt}\rrbracket_e^M = (\emptyset, \{s_0\})$
It follows that: $\llbracket \texttt{<}a\texttt{>tt} \vee \texttt{<-}a\texttt{>tt}\rrbracket_e^M = (\emptyset, \{s_0\})$
We conclude that they are no states in the model satisfying the formula ψ and that only the state s_0 might satisfy this formula. But, a deeper analysis of this formula shows that the state s_0 should satisfy it. In fact, this formula expresses the two following facts:

1. A state has an a-transition;

2. A state has an ”-a”-transition, i.e., an $\{b, c\}$-transition.

Combining these two facts, we conclude that a state satisfies this formula if it has an $\{a, b, c\}$-transition, i.e., has a transition labelled with any action since $\{a, b, c\} = \mathcal{A}ct$. Consequently, the state s_0 should satisfy the formula.

Table 5 shows the new definition of the semantics of the operators $\vee$ and $\wedge$.

Functions $\mathcal{R}_1$ and $\mathcal{R}_2$ are defined in Table 6.

Functions $\mathcal{E}_1$ and $\mathcal{E}_2$ are used respectively to extract from a disjunction and a conjunction the set of modal formulae of the form $\texttt{<}K_1\texttt{>}\,\psi_1$ and $\texttt{[}K_1\texttt{]}\,\psi_1$:

$$\mathcal{E}_1(\texttt{<}K_1\texttt{>}\,\psi_1 \vee \ldots \vee \texttt{<}K_n\texttt{>}\,\psi_n) = \{\texttt{<}K_1\texttt{>}\,\psi_1, \ldots, \texttt{<}K_n\texttt{>}\,\psi_n\}$$

$$\mathcal{E}_2(\texttt{[}K_1\texttt{]}\,\psi_1 \wedge \ldots \wedge \texttt{[}K_n\texttt{]}\,\psi_n) = \{\texttt{[}K_1\texttt{]}\,\psi_1, \ldots, \texttt{[}K_n\texttt{]}\,\psi_n\}$$

Concerning the tableau-based proof system, Table 7 and 8 show the rules for sequents $H \vdash s \in \psi_1 \vee \psi_2$ and $H \vdash s \approx \psi_1 \vee \psi_2$.

Notice that proofs (Finiteness, Soundness and Completeness) attached to this improved version of the semantics are quite similar to the ones of the original logic.

Table 5: New semantics for $\vee$ and $\wedge$.

$$
\begin{aligned}
[\![\psi_1 \vee \psi_2]\!]_e^M = (\; & \pi_1([\![\psi_1]\!]_e^M) \cup \pi_1([\![\psi_2]\!]_e^M) \cup \mathcal{R}_1(\psi_1,\psi_2),\\
& \pi_2([\![\psi_1]\!]_e^M) \cup \pi_2([\![\psi_2]\!]_e^M)\\
)\; &
\end{aligned}
$$

$$
\begin{aligned}
[\![\psi_1 \wedge \psi_2]\!]_e^M = (\; & \pi_1([\![\psi_1]\!]_e^M) \cap \pi_1([\![\psi_2]\!]_e^M),\\
& \pi_2([\![\psi_1]\!]_e^M) \cap \pi_2([\![\psi_2]\!]_e^M) \cap \mathcal{R}_2(\psi_1,\psi_2)\\
)\; &
\end{aligned}
$$

Table 6: Definition of functions $\mathcal{R}_1$ and $\mathcal{R}_2$.

$$
\begin{aligned}
\mathcal{R}_1(\psi,\psi') = \{s \mid (\exists s' \mid s \xrightarrow{\delta} s' :\; & \\
& (\exists E \mid E = \{<K_1>\psi_1,\ldots,<K_n>\psi_n\} \subseteq \mathcal{E}_1(\psi \vee \psi') :\\
& \quad \bigcup_{i=1}^{n} K_i = \mathcal{A}ct \wedge s' \in \bigcap_{i=1}^{n} \pi_1([\![\psi_i]\!]_e^M)))\\
\}\; &
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}_2(\psi,\psi') = \{s \mid (\forall s' \mid s \xrightarrow{\delta} s' :\; & \\
& (\forall E = \{[K_1]\,\psi_1,\ldots,[K_n]\,\psi_n\} \subseteq \mathcal{E}_2(\psi \wedge \psi') :\\
& \quad \bigcup_{i=1}^{n} K_i = \mathcal{A}ct \Rightarrow s' \in \bigcup_{i=1}^{n} \pi_2([\![\psi_i]\!]_e^M)))\\
\}\; &
\end{aligned}
$$

Table 7: Rules for $H \vdash s \in \psi_1 \vee \psi_2$.

(R2$_1$) $\dfrac{H \vdash s \in \psi_1 \vee \psi_2}{H \vdash s \in \psi_1}$ (R2$_2$) $\dfrac{H \vdash s \in \psi_1 \vee \psi_2}{H \vdash s \in \psi_2}$

(R2$_3$) $\dfrac{H \vdash s \in \psi_1 \vee \psi_2}{H \vdash s' \in \bigwedge_{i=1}^{n} \psi^i}\; C_3$

(R3) $\dfrac{H \vdash s \in \neg(\psi_1 \vee \psi_2)}{H \vdash s \in \neg\psi_1 \quad H \vdash s \in \neg\psi_2 \quad H \vdash s' \in \bigvee_{i=1}^{n} \neg\psi^i}\; C_3$

$C_3 : s' \in \{s' \mid s \xrightarrow{\delta} s'\} \wedge \{<K_1>\psi_1,\ldots,<K_n>\psi_n\} \subseteq \mathcal{E}_1(\psi_1 \vee \psi_2) \wedge \bigcup_{i=1}^{n} K_i = \mathcal{A}ct$

Table 8: Rules for $H \vdash s \approx \psi_1 \vee \psi_2$.

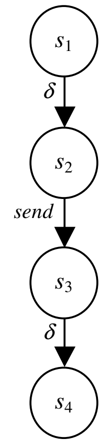
(R2'$_1$) $\dfrac{H \vdash s \approx \psi_1 \vee \psi_2}{H \vdash s \approx \psi_1}$ (R2'$_2$) $\dfrac{H \vdash s \approx \psi_1 \vee \psi_2}{H \vdash s \approx \psi_2}$

(R3') $\dfrac{H \vdash s \approx \neg(\psi_1 \vee \psi_2)}{H \vdash s \approx \neg\psi_1 \quad H \vdash s \approx \neg\psi_2}$

Table 9: C Code with it's corresponding abstract model.

```
extern int openFile(char*);
extern void closeFile(int);
extern int send(int, char*);

int main()
{
        int socket;
        int file;
        char fileName[1000];
        int (*f)(char *);
        ⋮
        f(”f1.ex”);
        send( socket, ”done” );
        f(”f2.ex”);
        return 0;
}
```



5 Extensions

In the previous sections, we have used models that admit δ-transitions abstracting unknown actions. Meanwhile, this abstraction is not adequate in some situations. We need more precise abstraction to obtain more results.

For instance, let's take the C code presented in Table 9 together with its associated abstract model, and let's take the following formula:

$$\text{eventually}(\,[\,\textit{openFile}\,]\,\text{eventually}(\text{do}(\textit{closeFile})))$$

that states that whenever a program does an *openFile* action, it will eventually do a *closeFile* action. Using the already defined semantics, we can conclude that the model described in Table 9 might satisfy this formula. In fact, the δ-transition would be first substituted by an *openFile* and then substituted by a *closeFile*. But, looking to the corresponding source code, one can deduce that the abstract actions δ correspond in fact to the same one. Hence, the model is not, in this case, a good abstraction of the program, and should not satisfy the formula.

The solution is to define, in the model, new transitions δ_i that should be instantiated only once. Hence, if a transition δ_i is instantiated with an action a, all the occurrences of δ_i in the model have to be instantiated with the same action. Figure 1 proposes a new model suitable for the C program described in Table 9.

Note that when a loop contains transitions δ, we distinguish one of the following two cases:

1. δ abstracts an invariant action. In this case, we use an δ_i action instead of the δ one.

2. δ could abstract, in each iteration, different actions. In this case, we use an δ action that could be instantiated differently.

In addition, let's take another C code presented in Table 10 together with its associated abstract model.

With regard to the previous formula, and using an adequate semantics, it is possible to attest that the model might satisfy this formula. Contrary to the previous example, the δ_2-transition could not be instantiated with any action in the model. Furthermore, if the model admits only actions in $\mathcal{A}ct = \{\textit{openFile,closeFile,send}\}$, and if the type of action *send* is $int \times char* \longrightarrow int$, we could conclude that the model satisfies the formula since only actions *openFile* and *closeFile* could instantiate abstract actions δ_1 and δ_2. The use of types allows to limit the number of possible instantiations associated to a given abstract action.

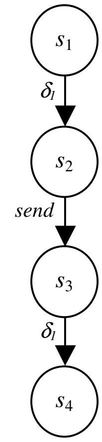


Figure 1: New Model with transitions δ_i.

6 Related Work

Despite their variety, existing model-checking techniques are typically limited to reasoning about action-based models and to using classical logics. However, in order to be able to reason under uncertainty, we need to adapt the semantics of logics.

Recently, a surge of interest has been devoted to the 3-valued logics. In [Yah01], the author presents a framework for verifying safety properties of concurrent Java programs using a 3-valued logic. His work takes place in the context of the dynamic allocation of Java thread objects and the verification of properties such as the absence of interference between threads. In [CEP01], the authors present an extension of classical CTL model-checking to reason about quasi-boolean multi-valued logics. More precisely, they use lattices to specify their logic and define multi-valued Kripke structures and multi-valued CTL. Furthermore, they describe a Java implementation of their symbolic model-checker, called $\mathcal{X}$Check. In [HJS01], the authors introduce Kripke Modal Transition Systems (Kripke MTSs) as a foundation for three-valued program analysis. Two applications of MTSs are presented to illustrate the power of their approach. More precisely, they show how MTSs are useful to reason about programs with uncertainty and how the logic for MTSs can express properties of practical interest.

7 Future Work

The previous sections propose a 3-valued logic to specify and verify security properties where only control flow aspects are considered. However, the information flow issue is of paramount importance. We must check that the program will not affect secrecy (by leaking sensitive information), integrity (by corrupting information), availability (by denying service to legal users), etc.

As future work, our intention is to extend the logic in a way which allows us to express more security properties. More specifically, our aim is to define properties involving data. The motivation is to track the information flow that could be a source of security violation. We intend to use variables to link information involved in different actions. We also intend to use types to abstract resources and labels to give an idea of the kind of information stored in each resource.

The model also has to be extended. Rather than keeping only sensitive actions, we have to propagate the security effects that result from accessing private resources, sending sensitive information over the network, etc. The main idea is to generate for each sensitive action, its effects on the resources and on the information flow.

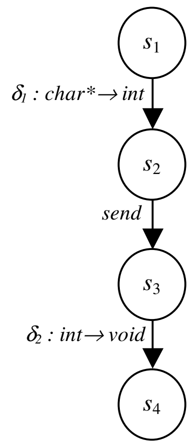
The aim of such work is to be able to specify and verify well-known security properties involving data. The following examples express such properties:

1. never(<$\texttt{read}(x, \rho_1^s)$>eventually(<$\texttt{write}(x, \rho_2^p)$>$\texttt{tt}$))
   This property states that no private information may be transferred to a public location. $\texttt{read}(x, \rho_1^s)$ represents the action of reading information, abstracted by variable x, from a resource of type ρ_1^s. The polymorphic type ρ_1 abstract any resources and the label s states that the information stored in the resource is *secret*. In the same

Table 10: C Code with it's corresponding abstract typed model.

```
extern int openFile(char*);
extern void closeFile(int);
extern int send(int, char*);

int main()
{
        int socket;
        int file;
        char fileName[1000];
        int (*f)(char*);
        void (*g)(int);
        ⋮
        file = f(”f1.ex”);
        send( socket, ”done” );
        g(file);
        return 0;
}
```



manner, $\texttt{write}(x, \rho_2^p)$ represents the action of writing information to a resource of type ρ_2^p. The label p stands for *public* information.

2. never($\texttt{<read}(\_, \mathit{file}^s)\texttt{>tt}$)
   This property states that no secret information may be read from any file.

3. never($\texttt{<read}(\_, \mathit{file}^\alpha)\texttt{>tt}$)
   This property is more restrictive since it prevents any access to files.

4. never($\texttt{<read}(f, \mathit{dir}^\alpha(\texttt{sysdir}))\texttt{>}$
   eventually($\texttt{<read}(x, \mathit{file}^\alpha(f))\texttt{>}$
   eventually($\texttt{<write}(x, \mathit{net}^p)\texttt{>tt}$)))

   This property states that no information coming from the system directory files may be sent on the public network.

5. always($\texttt{[write}(f, \mathit{dir}^\alpha(\texttt{tempdir}))\texttt{]}$
   eventually($\texttt{<delete}(\_, \mathit{file}^\alpha(f))\texttt{>tt}$))

   This property states that each file created in a temporary directory must be deleted.

Note that in the last two examples, variable f is used both to abstract an information and a particular resource. This allows to express very interesting properties.

8 Conclusion

This paper defines a new logic for the specification and the verification of high-level security properties. The syntax and the semantics of this logic are inspired from the μ-calculus one. However, its main particularity is its 3-valued semantics that allows to raison under uncertainly. The expressiveness and the usefulness of this logic have been proven via several varied examples. Actually, we are able to efficiently verify numerous interesting security properties. In addition, we have proposed some extensions, such as adding variables and types to model, in order to enhance the verification process.

As future works, we plan to extend the semantics in order to be able to capture properties related to flow of information and the use of resources as presented in Section 7. Finally, we intend to examine other interesting features of 3-valued modal logics, specially those in [HJS01], and implement them in our logic framework.

Bibliography

[BDD+99] Bergeron, J., M. Debbabi, J. Desharnais, B. Ktari, M. Salois and N. Tawbi, *Detection of malicious code in COTS software: A short survey*, in: *First International Software Assurance Certification Conference (ISACC'99)*, Washington D.C, 1999.

[CEP01] Chechik, M., S. Easterbrook and V. Petrovykh, *Model-checking over multi-valued logics*, in: J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity International Symposium of Formal Methods Europe, Berlin, Germany*, Lecture Notes in Computer Science 2021 (2001), pp. 72–98.

[Cle90] Cleaveland, R., *Tableau-based model checking in the propositional mu-calculus*, Acta Informatica **27** (1990), pp. 725–747.

[GS93] Gries, D. and F. B. Schneider, "A Logical Approach to Discrete Math," Texts and monographs in computer science, Springer, 1993.

[HJS01] Huth, M., R. Jagadeesan and D. Schmidt, *Modal transition systems: A foundation for three-valued program analysis*, in: *Proceedings of the European Symposium on Programming (ESOP2001)*, 2001, p. 15.

[Koz83] Kozen, D., *Results on the propositional mu-calculus*, Theoretical Computer Science **27** (1983), pp. 333–354.

[Koz98] Kozen, D., *Efficient code certification*, Technical Report TR98-1661, Cornell University (1998).

[Kta03] Ktari, B., "Certification de composantes logicielles," Ph.D. thesis, Laval University, Quebec, Canada (2003).

[MWCG99] Morrisett, G., D. Walker, K. Crary and N. Glew, *From system F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 528–569.

[Nec97] Necula, G. C., *Proof-carrying code*, in: N. D. Jones, editor, *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (1997), pp. 106–119.

[SW89] Stirling, C. and D. Walker, *Local model checking in the modal mu-calculus*, in: *Proceedings of TAPSOFT'89*, Lecture Notes in Computer Science 351 (1989), pp. 161–177.

[Yah01] Yahav, E., *Verifying safety properties of concurrent Java programs using 3-valued logic*, ACM SIGPLAN Notices **36** (2001), pp. 27–40.

Part II

Security by Construction

Domain Separation by Construction

William Harrison Mark Tullsen James Hook

Pacific Software Research Center
OGI School of Science & Engineering
Oregon Health & Science University
Beaverton, Oregon, USA
{wlh,mtullsen,hook}@cse.ogi.edu

Abstract

This paper advocates a novel approach to language-based security: by structuring software with monads (a form of abstract data type for effects), we are able to maintain separation of effects by construction. The thesis of this work is that well-understood properties of monads and monad transformers aid in the construction and verification of secure software. We introduce a formulation of non-interference based on monads rather than the typical trace-based formulation. Using this formulation, we prove a non-interference style property for a simple instance of our system model. Because monads may be easily and safely represented within any pure, higher-order, typed functional language, the resulting system models may be directly realized within such a language.

1 Introduction

This paper advocates a novel approach to language-based security: security by construction. Starting from a mathematical model of shared-state concurrency, we outline the development by stepwise refinements of an operating system kernel supporting both standard Unix-like system calls (e.g., fork, sleep, etc.) and a formally verified security policy (domain separation). Previous work has focused on languages with explicit or implicit imperative features, such as Java or ML; our approach assumes a purely functional language in which all imperative features are captured by monads. As a result all impure effects (references, exceptions, I/O) are distinguished from pure computations by their types, and thus side-effects are allowed while preserving the semantics of the purely functional subset of the language.

The research reported here presents a formal, language-based model of security combining three approaches to system security and language semantics:

- **Security "By Design."** Some approaches advocate implementation strategies for secure system construction, with the idea that such disciplined strategies are more likely to produce secure systems. One such strategy used in Java implementations, *sandboxing*, limits the scope of stateful effects by executing threads in disjoint regions of memory as illustrated in Figure 2. Good engineering, however, does not constitute a guarantee of any security policy.

- **Trace-based Formal Security Models.** There are a number of formal security models [GM90, McC88, ZL97, McL94, Rus82] which characterize permissible interactions between concurrent threads in terms of traces of abstract events. These models make precise the intuition that low-security operations should be oblivious to the execution of high-level operations. One feature of such models is that their precise relationship to actual systems remains unclear.

- **Monadic Language Semantics.** The approach advocated here combines "by design" security with trace-based models into a common framework based on *monads*. Monads provide a mathematically sophisticated theory of effects which has proven useful in denotational semantics [Mog90, Lia98, Pap01], functional programming [Wad92], and software verification [Har01]. Structuring our system specifications with monads yields many benefits, not the least of which are a number of useful properties obtained "by construction" which simplify the verification of our security property.

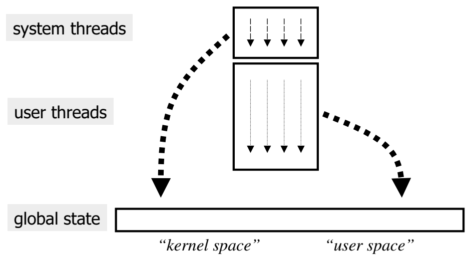


Figure 1: *Shared-state concurrency with global state*: Threads access the same global state, potentially interfering in ways difficult to control.

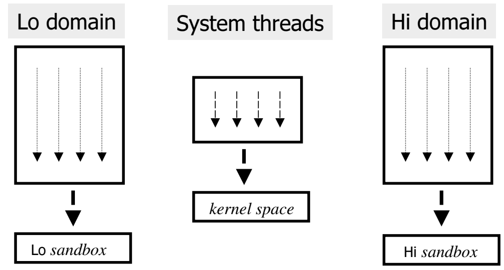


Figure 2: *Concurrency with Domain Separation*: Scoping of effects is achieved by partitioning the global state into separate sandboxes.

1.1 A Monadic Model for Security

We present a formal model of security in which the model itself may be refined to an implementation of a system with secure shared-state concurrency. Essential to our approach is the use of monads and monad transformers to structure our specifications. It is our thesis that systems constructed monadically are more easily verified because of the monadic encapsulation of effects. Monads and monad transformers allow us to reason about our system definitions at the level of denotational semantics. Because monads may be easily realized within any higher-order functional programming language, system specifications are readily executable.

Many formal security models are formulated in terms of sequences of abstract events. For the sake of convenience, we will refer to such models as *event systems*. The intended interpretation of events is that they are imperative operations on a shared state, but this is not made explicit—that is, the events themselves are uninterpreted. Our approach makes this interpretation explicit by considering languages of system behaviors (written *Beh*) and their denotational semantics. For the sake of simplicity, we assume there are only two separated domains, Hi and Lo, corresponding to two security levels of the same name. However, all of our results generalize easily to n separate domains.

According to our view of language-based security, an arbitrary, interleaved sequence of Lo and Hi operations in a traditional traced-based model:

$$h_0, l_0, \ldots, h_n, l_n$$

is viewed as the imperative *Beh*-program describing a particular (partial) system behavior:

$$h_0 \;;\; l_0 \;;\; \ldots \;;\; h_n \;;\; l_n$$

We then give such programs a (monadic) denotational semantics:

$$[\![-]\!] : (\mathit{Beh}_{\mathsf{Lo}} + \mathit{Beh}_{\mathsf{Hi}}) \to \mathsf{R}() \tag{1}$$

where R is a monad encapsulating imperative effects and a notion of interleaving concurrency called resumptions[Plo76, Mog90, Pap01]. The monad R is constructed especially to isolate Hi, Lo, and kernel effects from one another.

The security property we prove may be intuitively described as the execution of low security events being *oblivious* to the execution of high security events. For any initial sequence of interleaved Hi and Lo events

$$h_0 \;;\; l_0 \;;\; \ldots \;;\; h_n \;;\; l_n$$

the effect of its execution on the Lo state should be identical to that of executing the Lo events in isolation:

$$l_0 \;;\; \ldots \;;\; l_n$$

The precise definition of the security policy, using the above denotational semantics, is developed in Section 5.

1.2 Partitioning Effects with Monads

Domain separation is supported by partitioning the state into disjoint pieces, with each piece corresponding to a separate domain. Stateful operations are then given a security level and can only manipulate the storage partition corresponding to its security level. Achieving process separation via partitioning, sometimes referred to as *sandboxing*, seems to have originated in the work of Rushby [Rus82, Rus81]. With monads, it is a simple matter to partition storage into sandboxes, and this process is particularly straightforward when the monads are constructed with monad transformers.

How this partitioning works is illustrated in Figure 3. Corresponding to the security levels Hi and Lo are separate domains (marked **(c)**) that maintain distinct stores H and L, respectively. Hi and Lo stateful operations are then encapsulated within monads of the same name, created with the monad transformers (StateT H) and (StateT L), respectively. Hi and Lo stateful operations h and l may be executed by lifting them to the kernel level (marked **(b)**) with *liftH* and *liftL*, respectively, and these lift mappings are created by the application of the state monad transformers. Separation of effects is maintained by these lift mappings—and precisely how is described in detail in Section 4 below.

1.3 Separation By Construction

The approach advocated here achieves secure shared-state concurrency by construction, where "by construction" is used in two different, yet complementary, senses. The process is depicted in Figure 4, which illustrates both meanings of "by construction":

- **Stepwise-refinement of System.** The vertical axis of Figure 4 measures the richness of system behaviors, and each step along that axis marks an addition to those behaviors. At point 0, only a single, monolithic process domain exists and all thread scheduling is static. At point 1, threads are executable on separate domains. Point 2 allows statically-scheduled multitasking, while point 3 allows the scheduling of threads to occur dynamically. Dynamic scheduling is necessary for threads to affect their own execution behavior (as with the Unix system call `sleep` and intra-domain synchronization mechanisms such as semaphores) or to affect the system "wait list" (as with the Unix system call `fork`). Point 4 adds such thread-level control operations, and point 5 allows for secure, interdomain communication (such as asynchronous message-passing obeying a "no-write-down" security policy).

- **Properties of Monads & Monad Transformers.** Many of the above enhancements to system functionality are reflected in refinements to the underlying monads and monad transformers of Figure 3. Monads and monad transformers allow the effects of threads of differing security levels to be controlled in a mathematically rigorous manner, and this scoping of effects tames insecure interference between threads of differing security levels. The "by construction" properties of monads and monad transformers are extremely useful in formally demonstrating domain separation.

1.4 Overview

Section 2 surveys related work and Section 3 summarizes the necessary background on monads, monad transformers, and resumption based concurrency for understanding this work. Section 4 describes the useful properties that are obtained by construction with monad transformers. Section 5 presents the main results of this research—the stepwise development of an operating system kernel for secure computation. First, a language of system behaviors is defined and given a resumption monadic semantics in Section 5.1. Second, a system for shared-state concurrency with global store is transformed into a system with separated domains in Section 5.2. Third, security in this setting—*take separation*—is specified and verified in Section 5.3. Section 6 gives a brief overview of the development path from system 1 to system 5 in Figure 4. Finally, Section 7 summarizes the present work and outlines future directions.

2 Related Work

Formulating security policies in terms of non-interference goes back to the work of Goguen and Meseguer [GM90, GM84] and our notion of domain separation is based on theirs. Haigh and Young [HY86] and Rushby [Rus92] have extended this work to the intransitive case (where the "interferes" relation is not required to be transitive). Non-interference has been extended to concurrent [SV98, BC01] and probabilistic models [GI90].

Many formal security models—including non-interference and separability—are formulated in terms of *event systems* [GM90, McC88, ZL97, McL94]. While there are a variety of similar formulations of event systems, they all include the following: an event system S is comprised of a set of abstract *events* E and a set of *traces* T of (potentially infinite) sequences of events in E. Furthermore, each event is assigned a security level, which, for the sake of the present discussion, we will assume to be Lo and Hi[1].

[1] Some formulations also distinguish subsets of E as *input* and *output* events.

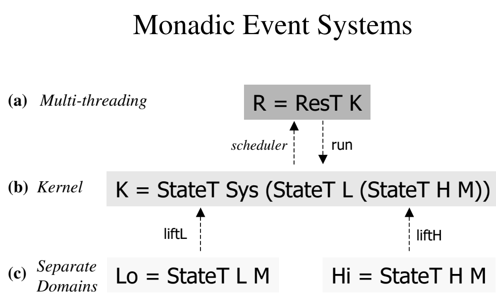


Figure 3: Observable events created in isolated security domains Lo and Hi are lifted to the kernel K. Laws about monad transformers (StateT and ResT) aid in proving domain separation.

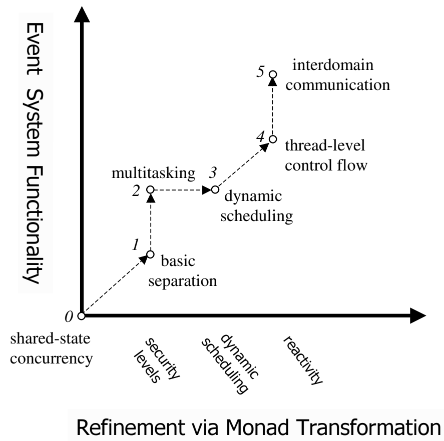


Figure 4: *Separation by construction*: System is enhanced through refinement of monad transformers ; "by construction" properties of monad transformers aid in verifying separation.

Security policies may then be formulated as relationships between event traces which specify permitted system executions. These typically are of the form: if $\tau_1, \ldots, \tau_n \in T$, then $f(\tau_1, \ldots, \tau_n) \in T$, where function f combines the traces τ_i in some manner specific to the security policy. For example, *separability* [McL94, ZL97] can be defined formally as: for every pair of traces $\tau_1, \tau_2 \in T$, then any trace τ such that $(\tau \downarrow_{\mathsf{Lo}}) = (\tau_1 \downarrow_{\mathsf{Lo}})$ and $(\tau \downarrow_{\mathsf{Hi}}) = (\tau_2 \downarrow_{\mathsf{Hi}})$ is also in T. Here $(\tau \downarrow_{\mathsf{Lo}})$ and $(\tau \downarrow_{\mathsf{Hi}})$ filter out all but the Lo and Hi events from τ, respectively. As formulated these techniques are easily generalized to any total order of security levels known statically.

The research reported here is inspired by Rushby's work [Rus82, Rus81] on abstract operating systems called *separation kernels*. A separation kernel enforces process isolation by partitioning the state into separate user spaces (called "colours"), allowing reasoning about the processes as if they were physically distributed. The security levels Hi and Lo used in the present work correspond to colours. Separation kernels are then specified using finite-state machines, and separability (i.e., that differently-colored processes do not interfere) is characterized in terms of event traces arising from executions of these machines.

In the context of the Programatica project at OGI we are working to develop and formally verify *OSKer* (Oregon Separation Kernel), a kernel for MLS applications. To formally verify security properties of such a system is a formidable task. One approach to reducing the enormity of this task has been to use type-systems for information-flow. There has been a growing emphasis on such *language-based* techniques for information flow security [VS97, SV98, HR98, Smi01, PS02] (see [SM03] for a survey of this work). The chief strength of this type-based approach is that the well-typedness of terms can be checked statically and automatically, yielding high assurance at low cost. Unfortunately, this type-based approach is not as general as one might wish: first, there will be programs which are secure but which will be rejected by the type system due to lack of precision, and second, there will be programs that have information flow leaks which we want to allow (e.g., a declassification program) but which would be rejected by the type system.

Moggi was the first to observe that the categorical structure known as a monad was appropriate for the development of modular semantic theories for programming languages [Mog90]. In his initial development, Moggi showed that most known semantic effects could be naturally expressed monadically. He also showed how a sequential theory of concurrency could be expressed in the resumption monad [Mog90]. That model of concurrency is used extensively in the present work. Wadler and colleagues at Glasgow University observed that using monads internally in the pure, higher-order, typed language Haskell gave a natural and safe embedding of effectful computation in a pure language [PJP93].

Modeling concurrency by resumptions was introduced by Plotkin [Plo76, Sch86]. Moggi showed how resumptions could be modeled with monads [Mog90] and our formulation of resumptions in terms of monad transformers is that of Papaspyrou [Pap01]. Although continuations can also model concurrency [Wan80, Cla99, FFKF99], resumptions can be viewed as a disciplined use of continuations which allows for simpler reasoning about our system.

There have been many previous attempts to develop secure OS kernels: PSOS [NBF$^+$80], KSOS [MD79], UCLA

Secure Unix [WKP80], SAT [BYKH85], KIT [Bev89], EROS [SSF99], and MASK [WMGT00] among others. There has also been work using functional languages to develop high confidence system software: the Fox project at CMU [HLP98] is a case in point of how typed functional languages can be used to build reliable system software (e.g., network protocols, active networks); the Ensemble project at Cornell [BCH+00] uses a functional language to build high performance networking software; and the Switchware project [AAH+98] at the University of Pennsylvania is developing an active network in which a key part of their system is the use of a typed functional language.

3 Background

Some familiarity with monads and their uses in the denotational semantics of languages with effects is essential to understanding the current work. We assume of necessity that the reader possesses such familiarity. This section serves as a quick review of the basic concepts of monads and monadic semantics, and readers requiring more should consult the references for further background. An informal introduction to monads can be found in Wadler [Wad92]. A formal treatment of monads and their rôle in denotational semantics is in Moggi [Mog90].

Monads [Mog90] can be understood as abstract data types (ADTs) for defining languages and programs. A monad ADT can encapsulate such language features as state, exceptions, multi-threading, environments, and CPS. Although combinations of such features can be encapsulated in a single monad, a more modular approach, in which each feature can be treated separately, is achieved with *monad transformers* [Mog90, LHJ95, Lia98]. Monad transformers allow us to easily combine and extend monads. A monad is extended similarly to how a class is extended using inheritance in object oriented languages: what makes sense in a monad "class" makes sense in the "subclass" created by inheritance. In this section we will introduce monads and monad transformers, we will then describe the resumption monad transformer.

3.1 Monads and Monad Transformers

A monad is a triple $\langle \mathsf{M}, \eta, \star \rangle$ consisting of a type constructor M and two operations:

$$\begin{array}{rcll} \eta & : & a \to \mathsf{M}\, a & \text{(unit)} \\ (\star) & : & \mathsf{M}\, a \to (a \to \mathsf{M}\, b) \to \mathsf{M}\, b & \text{(bind)} \end{array}$$

The η operator is the monadic analogue of the identity function, injecting a value into the monad. The $\star$ operator gives a form of sequential application. These operators must satisfy the monad laws:

$$\begin{array}{rcll} (\eta\ v) \star k & = & k\ v & \text{(left unit)} \\ x \star \eta & = & x & \text{(right unit)} \\ x \star (\lambda a.(k\ a \star h)) & = & (x \star k) \star h & \text{(assoc)} \end{array}$$

In what follows we often use the $\gg$ operator, defined in terms of $\star$ thus:

$$x \gg k \quad = \quad x \star \lambda\_.k$$

The lambda null binding (i.e., "$\lambda\_$") acts as a dummy variable, ignoring the value produced by x. The identity monad $\langle \mathsf{Id}, \eta_{\mathsf{Id}}, \star_{\mathsf{Id}} \rangle$ is defined as follows:

$$\mathsf{Id}\ a \ = \ a \qquad x \star_{\mathsf{Id}} k \ = \ k\, x \qquad \eta_{\mathsf{Id}}\ x \ = \ x$$

Another example is the state monad $\langle \mathsf{St}\ s, \eta_{\mathsf{St}}, \star_{\mathsf{St}} \rangle$ defined as follows:

$$\begin{array}{lcllcl} \mathsf{St}\ s\ a & = & s \to (a \times s) & \eta_{(\mathsf{St}\ s)}\ x & = & \lambda\sigma : s.(x, \sigma) \\ \mathsf{u}(\Delta) & = & \lambda\sigma.((), \Delta\ \sigma) & x \star_{(\mathsf{St}\ s)} f & = & \lambda\sigma_0.\ \text{let}\ (a, \sigma_1) = x\ \sigma_0\ \text{in}\ f\ a\ \sigma_1 \\ \mathsf{g} & = & \lambda\sigma.(\sigma, \sigma) & & & \end{array}$$

Here, () signifies both the unit type and the single element of that type. The operators u and g, for updating and getting the state respectively, are defined only for the $(\mathsf{St}\ s)$ monad. Such monad specific operators are referred to as non-proper morphisms.

There are various formulations of monad transformers, we follow that given in [LHJ95] where a monad transformer consists of a type constructor T, a mapping from a given monad $\langle \mathsf{M}, \eta_{\mathsf{M}}, \star_{\mathsf{M}} \rangle$ to a new monad $\mathsf{T}\ \mathsf{M} = \langle \mathsf{T}\ \mathsf{M}, \eta_{(\mathsf{T}\ \mathsf{M})}, \star_{(\mathsf{T}\ \mathsf{M})} \rangle$, and an associated function $\mathrm{lift}_{\mathsf{T}}$. The function $\mathrm{lift}_{\mathsf{T}} : \mathsf{M}\ a \to \mathsf{T}\ \mathsf{M}\ a$ performs a "lifting" of computations in M to computations in $(\mathsf{T}\ \mathsf{M})$; and will generally satisfy the *Lifting Laws* [Lia98]:

$$\mathrm{lift} \circ \eta_{\mathsf{M}} = \eta_{(\mathsf{T}\ \mathsf{M})} \qquad \mathrm{lift}(x \star_{\mathsf{M}} f) = (\mathrm{lift}\ x) \star_{(\mathsf{T}\ \mathsf{M})} (\mathrm{lift} \circ f)$$

These laws ensure that a monad transformer adds features without changing existing features of the base monad M.

As an example, the state monad $\mathsf{St}\,s$ can be written more generally as the monad transformer $\mathsf{StateT}\,s$ which transforms M (see Definition 3.1). Note that the monad transformer $\mathsf{StateT}\ s$ applied to the identity monad Id gives the original $\mathsf{St}\ s$ monad.

In Definitions 3.1 and 3.2, the unit and bind of the new monads are η and $\star$, respectively, while those of the transformed monad M are subscripted.

Definition 3.1 (State Monad Transformer) *For monad* M *and type* s,

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{M}'\,a = \mathsf{StateT}\,s\,\mathsf{M}\,a = s \to \mathsf{M}(a \times s) & & & u & : & (s \to s) \to \mathsf{M}'() \\
\mathrm{lift}\,x & = & \lambda\sigma.\,x \star_{\mathsf{M}} \lambda y.\,\eta_{\mathsf{M}}(y,\sigma) & u\,\Delta & = & \lambda\sigma.\,\eta_{\mathsf{M}}((),\Delta\,\sigma) \\
\eta\,x & = & \lambda\sigma.\,\eta_{\mathsf{M}}(x,\sigma) & g & : & \mathsf{M}'\,s \\
x \star f & = & \lambda\sigma_0.\,(x\,\sigma_0) \star_{\mathsf{M}} (\lambda(a,\sigma_1).f\,a\,\sigma_1) & g & = & \lambda\sigma.\,\eta_{\mathsf{M}}(\sigma,\sigma)
\end{array}$$

The following resumption monad transformer provides the foundation for the basic concurrency model here [Pap01], corresponding to point 0 in Figure 4. To avoid confusion, we label resumption monad transformers by their corresponding point number from Figure 4. Section 3.2 explains the intuitions behind this monad transformer.

Definition 3.2 (Resumption Monad Transformer) *For monad* M,

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{ResT}_0\,\mathsf{M}\,a = \mu\mathsf{R}.\mathrm{D}\,a + \mathrm{P}\,(\mathsf{M}(\mathsf{R}\,a)) & & & & & \\
\mathit{step}\,\phi & = & \mathrm{P}(\phi \star_{\mathsf{M}} \lambda v.\eta_{\mathsf{M}}(\mathrm{D}\,v)) & (\mathrm{D}\,v) \star f & = & f\,v \\
\eta\,x & = & \mathrm{D}\,x & (\mathrm{P}\,m) \star f & = & \mathrm{P}(m \star_{\mathsf{M}} \lambda r.\eta(r \star_{\mathsf{M}} f))
\end{array}$$

3.2 Resumption Based Concurrency

This section introduces concurrency based on the resumption monad transformer, the definition of which can be found in Definition 3.2. How resumption based concurrency works is best explained by an example. We define a *thread* to be a (possibly infinite) sequence of "atomic operations." We make this notion precise below, but for the moment, assume that an atomic operation is a single machine instruction and that a thread is a stream of such instructions characterizing program execution. Consider first that we have two simple threads $a = [a_0; a_1]$ and $b = [b_0]$. According to the "concurrency as interleaving" model, concurrent execution of threads a and b means the set of all their possible interleavings: $\{[a_0; a_1; b_0], [a_0; b_0; a_1], [b_0; a_0; a_1]\}$.

But how do computations in a resumption monad correspond to threads? If the atomic operations of a and b are computations of type $\mathsf{M}\,()$, then the computations of type $\mathsf{ResT}_0\ \mathsf{M}\,()$ are the set of possible interleavings:

$$\begin{array}{c}
P\,(a_0 \gg \eta(P\,(a_1 \gg \eta(P\,(b_0 \gg \eta(D\,()))))))\\
P\,(a_0 \gg \eta(P\,(b_0 \gg \eta(P\,(a_1 \gg \eta(D\,()))))))\\
P\,(b_0 \gg \eta(P\,(a_0 \gg \eta(P\,(a_1 \gg \eta(D\,()))))))
\end{array}$$

Closer comparison of both versions of these interleaving semantics reveals a strong similarity. While the stream version implicitly uses a lazy "cons" operation $(h : t)$, the monadic version[2] uses something similar: $P\,(h \gg \eta t)$. This is important because threads may be infinite, and the laziness of P allows infinite *computations* to be constructed in $(\mathsf{ResT}_0\,\mathsf{M})$ just as the laziness of cons in $(h : t)$ allows infinite *streams* to be constructed.

Finally, we note that the resumption semantics of concurrency involves the elaboration of all possible thread interleavings, and that, while such a semantics may be expressed monadically via the non-determinism monad [Mog90, LHJ95], it is not computationally tractable. We choose instead to pick out a single interleaving via a scheduler.

4 "By Construction" Properties

We get a number of useful properties by construction through the use of monad transformers. The state monad transformer's lift mappings have two principal uses here. First, lifting preserves stateful behavior. In Figure 3 for example, this means that Hi and Lo operations behave the same when lifted to the kernel monad K as at their respective base monads. This is formally captured in Section 4.1 below. Furthermore, liftings delimit the effects of stateful operations on separate domains; this phenomenon—which we call *atomic non-interference*—is described in detail in Section 4.2.

Atomic non-interference is not only useful for proving the security property in Section 5.3, but it captures *precisely* what we mean by *monadic scoping of effects*. The monadic structure of the system semantics is the foundation on which domain separation is built.

[2]Read P as "pause" and D as "done."

4.1 State Monads and Their Axiomatization

This section presents an algebraic characterization of state monads. Intuitively, a state monad is a monad with non-proper morphisms to manipulate state. The behavior of these non-proper morphisms is captured by axioms below. Not surprisingly, it is then demonstrated that the state monad transformer creates state monads and preserves existing state monads.

Definition 4.1 (State Monad Structure) *The quintuple* $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$ *is a* state monad structure *when* $\langle \mathsf{M}, \eta, \star\rangle$ *is a monad, and the* update *and* get *operations on* s *are:* $\mathsf{u} : (s \to s) \to \mathsf{M}()$ *and* $\mathsf{g} : \mathsf{M}\, s$.

We will refer to a state monad structure $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$ simply as M if the associated operations and state type are clear from context.

The following axiomatization of the state monad is not meant to be complete. Rather, it reflects the properties of state monads required later in the proofs.

Definition 4.2 (State Monad Axioms) *Let* M *be the state monad structure* $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$. M *is a* state monad *if the following equations hold for any* $f, f' : s \to s$,

$$
\begin{array}{rcll}
\mathsf{u}\, f \star \lambda\_.\mathsf{u}\, f' & = & \mathsf{u}\,(f' \circ f) & \text{(sequencing)} \\
\mathsf{g} \star \lambda\_.\mathsf{u}\, f & = & \mathsf{u}\, f & \text{(cancellation)}
\end{array}
$$

The (sequencing) axiom shows how updating by f and then updating by f' is the same as just updating by their composition $(f' \circ f)$. The (cancellation) axiom specifies that g operations whose results are ignored have no effect on the rest of the computation.

For state monad $\langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$, a consequence of (sequencing) we use later is:

$$
\mathsf{u}\, f \gg \mathrm{mask} \quad = \quad \mathrm{mask} \qquad \text{(clobber)}
$$

where mask is defined as: $\mathsf{u}\ (\lambda\_.\sigma_0)$ for some state σ_0.

Theorem 4.3 (StateT creates a state monad) *For monad* M, *let monad* $\mathsf{M}' = \mathsf{StateT}\, s\, \mathsf{M}$, *with non-proper morphisms* u *and* g *added by* $(\mathsf{StateT}\ s)$. *Then* $\langle \mathsf{M}', \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}, \mathsf{g}, s\rangle$ *is a state monad.*

Theorem 4.4 (StateT preserves stateful behavior) *For any state monad* $\mathsf{M} = \langle \mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, s\rangle$, *the structure* $\langle \mathsf{StateT}\ s\, \mathsf{M}, \eta', \star', \mathrm{lift} \circ u, \mathrm{lift}\, g, s\rangle$ *is also a state monad, where* η', $\star'$, *and* lift *are the monadic unit, bind, and lifting operations, respectively, defined by* $(\mathsf{StateT}\ s)$.

4.2 Formalizing Atomic Non-interference

The second "by construction" property of monad transformers relates to how their associated lift mappings delimit stateful effects in monads created from multiple applications of the state monad transformers (e.g., the kernel monad K in Figure 3).

Definition 4.5 (Atomic Non-interference) *For monad* M *with bind operation* $\star$, *define the atomic non-interference relation* $\# \subseteq \mathsf{M}\,() \times \mathsf{M}\,()$ *so that, for* $\varphi, \gamma : \mathsf{M}\,()$, $\varphi \# \gamma$ *holds if and only if the equation* $\varphi \gg \gamma = \gamma \gg \varphi$ *holds.*

Theorem 4.6 *Let* M *be the state monad* $\langle \mathsf{M}, \eta_{\mathsf{M}}, \star_{\mathsf{M}}, \mathsf{u}_A, \mathsf{g}_A, A\rangle$. *By Theorem 4.3,* $\mathsf{M}' = \langle \mathsf{StateT}\, B\, \mathsf{M}, \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}_B, \mathsf{g}_B, B\rangle$ *is also a state monad. Then, for all* $\delta_A : A \to A$ *and* $\delta_B : B \to B$, $(\mathsf{u}_B\, \delta_B) \#_{\mathsf{M}'} \mathrm{lift}(\mathsf{u}_A\, \delta_A)$ *holds.*

Theorem 4.7 *Let* M *be a monad with two operations,* $a : \mathsf{M}\,()$ *and* $b : \mathsf{M}\,()$ *such that* $a \#_{\mathsf{M}} b$. *Then,* $(\mathrm{lift}\, a) \#_{(\mathsf{T}\,\mathsf{M})} (\mathrm{lift}\, b)$ *where* T *is a monad transformer and* $\mathrm{lift} : \mathsf{M}\, a \to (\mathsf{T}\,\mathsf{M})\, a$.

Monad Hierarchy

$$
\begin{array}{lllllll}
\mathsf{G_L} & = & \mathsf{StateT}\ G\ \mathsf{M}, & \mathsf{u}_G & : & (G \to G) \to \mathsf{G_L}() \\
\mathit{liftG} & = & \mathit{id} & \mathsf{g}_G & : & \mathsf{G_L}\, G \\
\mathsf{K} & = & \mathsf{G_L}, & \mathit{stepG} & : & \mathsf{K}\, a \to \mathsf{R}\, a \\
\mathsf{R} & = & \mathsf{ResT_0}\ \mathsf{K} & \mathit{stepG}\ \varphi & = & P_{\mathsf{Lo}}\,(\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(D\ v))
\end{array}
$$

Semantics of $\mathsf{G_L}$ Language

$$
\begin{array}{lll}
\multicolumn{3}{l}{ev\ :\ Beh \to \mathsf{R}\,()} \\
ev\,(x\,\texttt{:=}e) & = & (exp\ e) \star_{\mathsf{R}} \lambda v.\ (load \circ \mathsf{u}_G)[x \mapsto v] \\
ev\,(e_1\,\texttt{;}\,e_2) & = & (ev\ e_1) \gg_{\mathsf{R}} (ev\ e_2) \\
ev\ \texttt{skip} & = & (load \circ \mathsf{u}_G)\ (\lambda i.i) \\
ev\,(\texttt{ite}\ b\ e_1\ e_2) & = & exp\ b \star_{\mathsf{R}} \lambda v.\ if\ v = 0\ then\ (ev\ e_1)\ else\ (ev\ e_2) \\
ev\,(\texttt{while}\ \ b\ \texttt{do}\ c) & = & mwhile\ (exp\ b)\ (ev\ c) \\
mwhile\ b\ \varphi & = & b \star_{\mathsf{R}} \lambda v.\ if\ v = 0\quad then\quad \varphi \gg_{\mathsf{R}} (mwhile\ b\ \varphi) \\
& & \qquad\qquad\qquad\quad else\quad (load \circ \mathsf{u})\ (\lambda i.i)
\end{array}
$$

$$
\begin{array}{lllllll}
\multicolumn{3}{l}{exp : Exp \to \mathsf{R}\ Int} & & & & \\
exp\ x & = & (load\ \mathsf{g}_G) \star_{\mathsf{R}} \lambda\sigma.\eta_{\mathsf{R}}(\sigma\ x) & & load : \mathsf{G_L}\ a \to \mathsf{R}\ a & & \\
exp\ i & = & \eta_{\mathsf{R}}\ i & & load & = & step \circ \mathrm{lift}
\end{array}
$$

Figure 5: Shared-state Concurrency with Global State (system 0).

5 Basis for a Secure OS Kernel

In monadic event systems, behaviors are programs in the *Beh* language, and traces of events are the denotations of these programs according to a resumption monadic semantics. The abstract syntax and resumption semantics of the language of behaviors, *Beh*, are presented below. The *Beh* language contains sufficient expressiveness to allow for potentially infinite streams of operations because it includes loops. Importantly, it allows for expression of potentially interfering programs as well.

Definition 5.1 (Abstract Syntax for *Beh***)** *Below is a BNF syntax for Beh:*

$$
\begin{array}{lll}
Beh & ::= & Var\,\texttt{:=}Exp \mid \texttt{skip} \mid Beh\texttt{;}Beh \mid \\
& & \texttt{ite}\ Exp\ Beh\ Beh \mid \texttt{while}\ \ Exp\ \texttt{do}\ Beh \\
Exp & ::= & Var \mid Int
\end{array}
$$

5.1 Global Shared-State Concurrency

In this section, the most basic model for shared-state concurrency is constructed. The monadic event system described in this section corresponds to the point labelled with 0 in Figure 4.

Using the monad transformers $\mathsf{ResT_0}$ and StateT (defined in Section 3), we define, for any monad M, the monad hierarchy for shared-state concurrency with global state. This construction is summarized in Figure 5. Unlike the system pictured in Figure 3, this monadic event system does not provide separation, and so it has only one global domain (called $\mathsf{G_L}$ for "global"). In Figure 5, state type G is $Name \to Int$.

Associated with these monad constructions are a number of non-proper morphisms. The morphism u_G applies a state-to-state map to the current G state in the $\mathsf{G_L}$ monad, while the morphism g_G reads and returns the current G state. The u_G and g_G morphisms are defined by an application of the StateT monad transformer. The lifting *liftG* reinterprets $\mathsf{G_L}$ computations in the kernel monad K, and because the kernel monad K is just $\mathsf{G_L}$ in this example, *liftG* is merely identity. The morphism *step* creates a "paused" K computation and results from the application of the $\mathsf{ResT_0}$ transformer.

The resumption monadic denotational semantics for system behaviors, *ev*, is presented in Figure 5. It is just what one would expect given recent work on the resumption monadic semantics of concurrency [Pap01]. Please note that because each observable event (namely u or g) is wrapped by a *load*, each such event is paused with P. The *run*

Monad Hierarchy & Lo Morphisms

$$
\begin{array}{lcl}
\mathsf{Hi} & = & \mathsf{StateT}\ H\ \mathsf{M} \\
\mathsf{Lo} & = & \mathsf{StateT}\ L\ \mathsf{M} \\
\mathsf{K} & = & \mathsf{StateT}\ H\ (\mathsf{StateT}\ L\ \mathsf{M}) \\
\mathsf{R} & = & \mathsf{ResT}_1\ \mathsf{K}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{u}_L & : & (L \to L) \to \mathsf{Lo}() \\
\mathsf{g}_L & : & \mathsf{Lo}\, L \\
\mathit{liftL} & : & \mathsf{Lo}\, a \to \mathsf{K}\, a \\
\mathit{stepL} & : & \mathsf{K}\, a \to \mathsf{R}\, a \\
\mathit{stepL}\ \varphi & = & P_{\mathsf{Lo}}\,(\varphi \star_{\mathsf{K}} \lambda v.\eta_{\mathsf{K}}(D\, v))
\end{array}
$$

Semantics of Lo Language

$$
\begin{array}{lcl}
\mathit{evL}\ ::\ \mathit{Beh} \to \mathsf{R}\,() & & \\
\mathit{evL}\,(x\,\texttt{:=}e) & = & (\mathit{expL}\ e) \star_{\mathsf{R}} \lambda v.\ (\mathit{loadL} \circ \mathsf{u}_L)[x \mapsto v] \\
\mathit{evL}\,(e_1\,\texttt{;}\,e_2) & = & (\mathit{evL}\, e_1) \gg_{\mathsf{R}} (\mathit{evL}\, e_2) \\
\mathit{evL}\,\texttt{skip} & = & (\mathit{loadL} \circ \mathsf{u}_L)\,(\lambda i.i) \\
\mathit{evL}\,(\texttt{ite}\, b\, e_1\, e_2) & = & \mathit{expL}\, b \star_{\mathsf{R}} \lambda v.\ \mathit{if}\ v = 0\ \mathit{then}\ (\mathit{evL}\, e_1)\ \mathit{else}\ (\mathit{evL}\, e_2) \\
\mathit{evL}\,(\texttt{while}\ b\ \texttt{do}\ c) & = & \mathit{mwhile}\,(\mathit{expL}\, b)\,(\mathit{evL}\, c) \\
\mathit{mwhile}\ b\ \varphi & = & b \star_{\mathsf{R}} \lambda v.\ \mathit{if}\ v = 0\quad \mathit{then}\quad \varphi \gg_{\mathsf{R}} (\mathit{mwhile}\ b\ \varphi) \\
& & \qquad\qquad\qquad\quad\ \mathit{else}\quad (\mathit{loadL} \circ \mathsf{u}_L)\,(\lambda i.i)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{expL} : \mathit{Exp} \to \mathsf{R}\,\mathit{Int} & & \\
\mathit{expL}\ x & = & (\mathit{loadL}\ \mathsf{g}_L) \star_{\mathsf{R}} \lambda\sigma.\eta_{\mathsf{R}}(\sigma\ x) \\
\mathit{expL}\ i & = & \eta_{\mathsf{R}}\ i
\end{array}
\qquad
\begin{array}{lcl}
\mathit{loadL} : \mathsf{Lo}\ a \to \mathsf{R}\ a & & \\
\mathit{loadL} & = & \mathit{stepL} \circ \mathit{liftL}
\end{array}
$$

Figure 6: Shared-State Concurrency with Basic Separation (system 1). The *evH* semantics (not shown) is obtained from *evL* by replacing all "L"-suffixed operations (e.g., "u_L") by their corresponding "H"-suffixed operations (e.g., "u_H"). There are also morphisms u_H, g_H, *liftH*, *stepH*, and *loadH* as well.

morphism for this monadic event system, which projects resumption computations in R to the runtime platform K (see Figure 3), is defined as:

$$
\begin{array}{lcl}
\mathit{run}\ ::\ \mathsf{R}\, a \to \mathsf{K}\, a & & \\
\mathit{run}\,(D\, v) & = & \eta_{\mathsf{K}} v \\
\mathit{run}\,(P\, \varphi) & = & \varphi \star_{\mathsf{K}} \mathit{run}
\end{array}
$$

5.2 Basic Separation System

Event systems contain trace projections based on security level—we write these as "$\downarrow_{\mathsf{Hi}}$" and "$\downarrow_{\mathsf{Lo}}$." Monadic event systems need a similar capability, which is achieved by refining the resumption monad transformer of Papaspyrou [Pap01] to reflect the Hi and Lo security levels. The refined resumption monad transformer is:

$$\mathsf{ResT}_1\ \mathsf{M}\ a \quad = \quad \mu\mathsf{R}.D\ a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\, a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\, a))$$

The unit η is *D*, and the bind $\star$ of the transformed monad is defined just as one would expect:

$$
\begin{array}{lcl}
(D\, v) \star f & = & f\ v \\
(P_{\mathsf{Lo}}\, \varphi) \star f & = & P_{\mathsf{Lo}}\,(\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f)) \\
(P_{\mathsf{Hi}}\, \varphi) \star f & = & P_{\mathsf{Hi}}\,(\varphi \star_{\mathsf{M}} \lambda r.\ \eta_{\mathsf{M}}(r \star f))
\end{array}
$$

Using this refined resumption transformer and StateT, we define the new monad hierarchy and monadic event system in Figure 6 for any monad M. Here, L and H are state types equal to *Name* → *Int*. Associated with these monad constructions are a number of non-proper morphisms also shown in Figure 6. The morphisms u_L and u_H apply a state-to-state map to the current state in their respective monads, while the morphisms g_L and g_H read and return the current state. The liftings *liftL* and *liftH* reinterpret Lo and Hi computations, resp., in the kernel monad K. The aforementioned morphisms are all defined by applications of the StateT monad transformer. The morphisms *stepH* and *stepL* create a "paused" K computation in either the Hi or Lo security levels. The "step" functions result from the application of the ResT_1 transformer.

By Theorems 4.3 and 4.4, we know that the monad K is a state monad with the operations lifted from the Hi and Lo monads. That is, the following are state monads:

$$\langle \mathsf{K}, \eta_{\mathsf{K}}, \star_{\mathsf{K}}, (\mathit{liftL} \circ \mathsf{u}_L), (\mathit{liftL}\, \mathsf{g}_L), L\rangle \qquad \langle \mathsf{K}, \eta_{\mathsf{K}}, \star_{\mathsf{K}}, (\mathit{liftH} \circ \mathsf{u}_H), (\mathit{liftH}\, \mathsf{g}_H), H\rangle$$

There are two semantics for *Beh* corresponding to the Hi and Lo security levels—these are *evH* and *evL*, respectively. The following example illustrates how the monadic structure is key to making this approach work. The semantics *evL* (*evH*) creates traces by injecting the Lo (Hi) operations into the P_{Lo} (P_{Hi}) side of R. Let c be $(x := 1)$, then the low- and high-security meanings of c are:

$$\begin{array}{lcll} \mathit{evL}\, c & = & P_{\mathsf{Lo}}\, (\mathsf{u}_L[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(D\,())) & \text{(low)} \\ \mathit{evH}\, c & = & P_{\mathsf{Hi}}\, (\mathsf{u}_H[x \mapsto 1] \gg_{\mathsf{K}} \eta_{\mathsf{K}}(D\,())) & \text{(high)} \end{array}$$

Note that the assignment in c is mapped by *evL* and *evH* into lifted operations on *different* monads (i.e., Lo and Hi in (low) and (high), resp.). Then, this security assignment is maintained in the resumption trace by the P_{Lo} and P_{Hi} tags. Because u_L and u_H operate on different states, these assignments can not interfere (in the sense of Section 4.2).

Below are two schedulers for the basic separation model, *withHi* and *withoutHi*, for a simple monadic event system. The schedule $(\mathit{withHi}\ \mathit{lo}\ \mathit{hi})$ creates Lo and Hi threads from event behaviors lo and hi, resp., in a round-robin fashion. Schedule $(\mathit{withoutHi}\ \mathit{lo})$ creates a single Lo thread.

$$\begin{array}{ll} \mathit{withHi} \,:\, \mathit{Beh} \to \mathit{Beh} \to \mathsf{R}\,() & \mathit{withoutHi} \,:\, \mathit{Beh} \to \mathsf{R}\,() \\ \mathit{withHi}\ \mathit{lo}\ \mathit{hi} \,=\, \mathit{weave}\ (\mathit{evL}\ \mathit{lo})\ (\mathit{evH}\ \mathit{hi}) & \mathit{withoutHi}\ \mathit{lo} \,=\, \mathit{evL}\ \mathit{lo} \end{array}$$

$$\begin{array}{l} \mathit{weave} \,:\, \mathsf{R}\,() \to \mathsf{R}\,() \to \mathsf{R}\,() \\ \mathit{weave}\ (P_{\mathsf{Lo}}\, \varphi)\ (P_{\mathsf{Hi}}\, \gamma) = P_{\mathsf{Lo}}\, (\varphi \star_{\mathsf{K}} \lambda r_{lo}.\ \eta_{\mathsf{K}}(P_{\mathsf{Hi}}\, (\gamma \star_{\mathsf{K}} \lambda r_{hi}.\eta_{\mathsf{K}}(\mathit{weave}\ r_{lo}\ r_{hi})))) \end{array}$$

The *run* morphism from Figure 3, which projects resumption computations in R to the runtime platform K, is defined as:

$$\begin{array}{lcl} \mathit{run} \,::\, \mathsf{R}\,a \to \mathsf{K}\,a & & \\ \mathit{run}\ (D\ v) & = & \eta_{\mathsf{K}} v \\ \mathit{run}\ (P_{\mathsf{Lo}}\, \varphi) & = & \varphi \star_{\mathsf{K}} \mathit{run} \\ \mathit{run}\ (P_{\mathsf{Hi}}\, \varphi) & = & \varphi \star_{\mathsf{K}} \mathit{run} \end{array}$$

5.3 Security for this setting: take separation

This section develops a non-interference style specification of domain separation for monadic event systems. The question we answer is: given a resumption computation representing a schedule of threads on separated domains, how do we specify that those processes do not interfere? The answer we provide in this section adapts a technique for proving properties of streams to the resumption monadic setting.

A common technique for proving a property of infinite lists is to show that the property holds of all finite approximations (i.e., finite initial prefixes) of the list. A well-known version of this technique is the *take lemma* [Bir98]:

$$s_1 = s_2 \quad \Leftrightarrow \quad \forall\, (n < \omega).\ \mathit{take}\ n\ s_1 = \mathit{take}\ n\ s_2$$

where $(\mathit{take}\ n\ [v_1, \ldots, v_n, \ldots]) = [v_1, \ldots, v_n]$. To show two streams s_1 and s_2 are equal using the take lemma, one shows that, for any non-negative integer n, each length n prefix of s_1 and s_2 are equal.

The security property proved here is analogous to the take lemma—hence its name. If, for any initial sequence of interleaved Hi and Lo events with n Lo events obtained from a system execution:

$$h_0 \,;\, l_0 \,;\, \ldots ;\, h_n \,;\, l_n$$

the effect of its execution on the Lo state should be identical to that of executing the Lo events in isolation:

$$l_0 \,;\, \ldots ;\, l_n$$

Using the denotational semantics, we make this precise:

$$\mathit{run}\, [\![h_0 \,;\, l_0 \,;\, \ldots ;\, h_n \,;\, l_n]\!] \gg_{\mathsf{K}} \mathit{mask} = \mathit{run}\, [\![l_0 \,;\, \ldots ;\, l_n]\!] \gg_{\mathsf{K}} \mathit{mask}$$

Dynamic Scheduling (3)

$$\begin{array}{lcl} \mathsf{R} & = & \mathsf{ResT_3\ K} \\ \mathsf{K} & = & \mathsf{StateT}\ \mathit{Sys}\ (\mathsf{StateT}\ H\ (\mathsf{StateT}\ L\ \mathsf{Id})) \\ \mathit{Sys} & = & [\mathsf{R}()] \times [\mathsf{R}()] \end{array}$$

Thread-level Control (4)

$$\begin{array}{lcl} \mathsf{R} & = & \mathsf{ResT_3\ K} \\ \mathsf{K} & = & \ldots \\ \mathit{Sys} & = & [\mathsf{Re}\mathit{Rsp}] \times [\mathsf{Re}\mathit{Rsp}] \\ \mathsf{Re} & = & \mathsf{ResT_4\ K} \\ \mathit{Req}_4 & = & \mathit{Continue} + \mathit{Sleep} \\ \mathit{Rsp} & = & () \end{array}$$

Interdomain Communication (5)

$$\begin{array}{lcl} \mathsf{R} & = & \mathsf{ResT_3\ K} \\ \mathsf{K} & = & \ldots \\ \mathit{Sys} & = & [\mathsf{Re}\mathit{Rsp}] \times [\mathsf{Re}\mathit{Rsp}] \times [Int] \times [Int] \\ \mathsf{Re} & = & \mathsf{ResT_4\ K} \\ \mathit{Req}_5 & = & \mathit{Req}_4 + \mathit{Snd(Dom,Int)} + \mathit{Rcv(Dom,Var)} \\ \mathit{Rsp} & = & () \\ \mathit{Dom} & = & \mathit{Hi} + \mathit{Lo} \end{array}$$

Summary of Refinements to ResT

$$\begin{array}{lcl} \mathsf{ResT_0\ M}\ a & = & \mu\mathsf{R}.D\ a + P\,(\mathsf{M}(\mathsf{R}\ a)) \\ \mathsf{ResT_1\ M}\ a & = & \mu\mathsf{R}.D\ a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\ a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\ a)) \\ \mathsf{ResT_3\ M}\ a & = & \mu\mathsf{R}.D\ a + P_{\mathsf{Lo}}\,(\mathsf{M}(\mathsf{R}\ a)) + P_{\mathsf{Hi}}\,(\mathsf{M}(\mathsf{R}\ a)) + P_{\mathsf{K}}\,(\mathsf{M}(\mathsf{R}\ a)) \\ \mathsf{ResT_4\ M}\ a & = & \mu\mathsf{R}.\mathit{Done}\ a + \mathit{PauseLo}\,(\mathit{Req} \times (\mathit{Rsp} \to \mathsf{M}(\mathsf{R}\ a))) + \mathit{PauseHi}\,(\mathit{Req} \times (\mathit{Rsp} \to \mathsf{M}(\mathsf{R}\ a))) + \mathit{PauseK}\,(\mathsf{M}(\mathsf{R}\ a)) \end{array}$$

Figure 7: The rest of the story (Points 3-5 in Figure 4). This briefly summarizes the refinements to the monad hierarchies underlying these systems. We omit point 2 (multitasking) as it requires no refinement to the monad transformers.

Here, $[\![-]\!]$ is defined as in Equation 1 with *evL* and *evH* from Figure 6, and *mask* is a stateful operation on the K monad which overwrites all non-Lo states. The particular definition of *mask* differs as the monadic event system is refined, but here, it is merely $\mathit{liftHi}(\mathsf{u}_H\ (\lambda\_.h_0))$. The operator *mask* plays the rôle of $\downarrow_{\mathsf{Lo}}$ in a trace-based event system. We define an additional helper function, *takeLo*. The function (*takeLo* n) picks out the initial sequences containing n Lo events:

$$\begin{array}{lcl} takeLo : Int \to \mathsf{R}() \to \mathsf{R}() \\ takeLo\ 0\ x & = & D() \\ takeLo\ n\ (P_{\mathsf{Lo}}\ \varphi) & = & P_{\mathsf{Lo}}\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ (n-1)))) \\ takeLo\ n\ (P_{\mathsf{Hi}}\ \varphi) & = & P_{\mathsf{Hi}}\ (\varphi \star_{\mathsf{K}} (\eta_{\mathsf{K}} \circ (takeLo\ n))) \end{array}$$

We may now formulate and prove take separation for the Basic Separation system described in Section 5.2:

Theorem 5.2 (Take Separation) *Let lo and hi be any Beh programs, then for all natural numbers n,*

$$\begin{array}{ll} run\ (takeLo\ n\ (withoutHi\ (evL\ lo))) \gg mask \\ \quad = \quad run\ (takeLo\ n\ (\mathit{withHi}\ (evL\ lo)\ (evH\ hi))) \gg mask \end{array}$$

where $mask = \mathit{liftHi}(\mathsf{u}_H\ (\lambda\_.h_0))$ for any arbitrary fixed $h_0 \in H$.

Proof (sketch) of Theorem 5.2 by induction on n. All binds $\star$ and units η below are in the K monad. For any natural number n, the computation:

$$run\ (takeLo\ n\ (\mathit{withHi}\ (evL\ lo)\ (evH\ hi)))$$

will have the form: $(l_1 \gg h_1 \gg \ldots \gg l_n \gg h_n)$ for K-computations l_i and h_i. Note that l_i and h_i are not arbitrary K-computations, but rather are lifted update and get events from the Lo and Hi domains, respectively. We know this from the definitions of the semantics *evL* and *evH*. This permits reasoning about system behaviors using atomic noninterference, the state monad axioms, and the clobber rule in the following inductive proof.

$$\begin{array}{lll} \underline{\text{Case } n = 0.} \\ run\ (takeLo\ 0\ (withoutHi\ (evL\ lo))) \gg mask \\ \quad = \quad run(Done\ ()) \gg mask & & \{\text{defn takeLo}\} \\ \quad = \quad run\ (takeLo\ 0\ (\mathit{withHi}\ (evL\ lo)\ (evH\ hi))) \gg mask \\ \underline{\text{Case } n = k+1.} \\ run\ (takeLo\ (k+1)\ (\mathit{withHi}\ (evL\ lo)\ (evH\ hi))) \gg mask \\ \quad = \quad (l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg \eta()) \gg mask & & \{\text{right unit \& assoc.}\} \\ \quad = \quad l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg h_{(k+1)} \gg mask & & \{\text{clobber/cancellation}\} \\ \quad = \quad l_1 \gg h_1 \gg \ldots \gg l_{(k+1)} \gg mask & & \{l_{(k+1)} \# mask\} \\ \quad = \quad l_1 \gg h_1 \gg \ldots \gg mask \gg l_{(k+1)} & & \{\text{ind. hyp.}\} \\ \quad = \quad \underbrace{l_1 \gg \ldots}_{h_i \text{ excised}} \gg mask \gg l_{(k+1)} & & \{l_i \# mask\} \\ \quad = \quad l_1 \gg \ldots \gg l_{(k+1)} \gg mask & & \{l_{(k+1)} \# mask\} \\ \quad = \quad run\ (takeLo\ (k+1)\ (withoutHi\ (evL\ lo))) \gg mask \end{array}$$

6 The rest of the spectrum

Figure 7 summarizes the refinements to the monadic hierarchy along the development path in Figure 4—this includes points 3-5. While lack of space allows only a very high-level description of these systems, it is astonishing how easily system behavior may be enriched with only very small changes to the underlying monad hierarchy.

Point (3) extends system behavior with dynamic dispatch of threads. The *Sys* state now maintains wait lists for Hi and Lo threads. Kernel-level events (i.e., scheduling decisions) now are included in the ResT_3 transformer in the form of the P_{K} constructor. Point (4) extends system behavior with thread-level control flow in the form of a Unix-like "sleep" signal. Monad transformer ResT_4 is a refinement to ResT_3 which adds signals. ResT_4 provides a non-proper morphism, $usersignal : Req \to \mathsf{Re}\,Rsp$ allowing user threads to contact the kernel with requests (e.g., *Sleep*), and, due to the dynamic dispatch capability, the kernel may service such requests. Point (5) implements asynchronous send and receive by enriching the *Sys* state with Hi and Lo message buffers and the *Req* type with send and receive signals. A thread contacts the kernel with *usersignal* as before, and it is the responsibility of the kernel to ignore insecure, "write-down" requests (e.g., $usersignal(Snd(\mathsf{Lo}, msg))$) from Hi threads.

7 Conclusion

Although the verification of OSKer's security properties is not complete, the challenges of this verification have inspired the work here. Our approach is based on the assumption that fully-automated methods—such as information-flow type systems—are not currently powerful and general enough to verify a large, complex system such as OSKer. The formidable task of verifying OSKer can be kept tractable by these techniques: (1) the system is built using a purely functional, statically-typed, polymorphic language, and thus the majority of the system components can be reasoned about using only their types; and (2) the system is structured to minimize the code that must be reasoned about explicitly.

Our system structure is based on monads, originally introduced to allow principled, effectful programming in pure, higher-order functional programming languages. Using monads, imperative features can be added to a pure language while preserving the semantics of the purely functional subset of the language. This work demonstrates that the monadic approach to effects has benefits far beyond the mere imitation of imperative style: using monads we can precisely manage and control information-flow and scope of effects. How such control might be achieved in an imperative language remains an open question.

The implementation language must be typed and purely functional to ensure that the monad abstraction is not compromised. Though we use Haskell as our implementation language, we do not use its laziness or type classes in any essential way; thus our techniques could be applied using the pure subset of ML. We have implemented systems 0-5 from Figure 4 in Haskell, and these implementations are available by request from the first author.

Monad transformers have proven their usefulness for modularizing interpreters and compilers [Lia98, Har01], resulting in modular components from which systems can be created. In this work we have shown their usefulness for modularizing a very different type of system. As seen in Section 5, this modularity enables us to straightforwardly extend and refine our system to have greater functionality. Our formulation also allows for extending our notion of interference to the intransitive case [HY86, Rus92], thus allowing for trusted processes (such as cryptographic servers) that are allowed to reclassify information.

A modular system not only allows for easier extension but also results in more modular verification. As we extend the system, we extend and modify the verification proofs; we need not perform verification from scratch. Although this is a work in progress, we expect our techniques to scale as our system grows in functionality. Scalable techniques for the modeling and implementation of complex systems requiring high confidence in their security properties remains a grand challenge of computer science. We believe that the ultimate solution to this challenge will draw heavily from the theory and practice of purely-functional, higher-order, typed languages.

Bibliography

[AAH+98] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith. The switchware active network architecture. *IEEE Network*, May/June 1998.

[BC01] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. *Lecture Notes in Computer Science*, 2076:382+, 2001.

[BCH+00] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, January 2000.

[Bev89] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[Bir98] Richard J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

[BYKH85] W.E. Boebert, W.D. Young, R.Y. Kain, and S.A. Hansohn. Secure ada target: Issues, system design, and verification. In *Proc. IEEE Symposium on Security and Privacy*, pages 176–183, 1985.

[Cla99] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[FFKF99] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *International Conference on Functional Programming*, pages 138–147, 1999.

[GI90] J.W. Gray III. Probabilistic interference. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 170–179, 1990.

[GM84] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[GM90] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20, Los Alamitos, Ca., USA, April 1990. IEEE Computer Society Press.

[Har01] William Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[HLP98] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1998. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[HY86] J.T. Haigh and W.D. Young. Extending the non-interference version of MLS for SAT. In *Proc. IEEE Symposium on Security and Privacy*, pages 232–239, 1986.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[Lia98] Sheng Liang. *Modular Monadic Semantics and Compilation*. PhD thesis, Yale University, 1998.

[McC88] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–187, 1988.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[MD79] E. J. McCauley and P. J. Drongowski. KSOS—the design of a secure operating system. In *Proc. AFIPS National Computer Conference*, volume 48, pages 345–353, 1979.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

[NBF+80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proof. Technical Report CSL-116, SRI, May 1980.

[Pap01] Nikos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Crete, 2001.

[PJP93] SL. Peyton Jones and Wadler P. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, January 1993.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3), 1976.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, January 2002.

[Rus81] J. Rushby. Design and verification of secure systems. In *Proceedings* 8^{th} *Symposium on Operating System Principles*, pages 12–21, Pacific Grove, CA, 1981.

[Rus82] J. Rushby. Proof of separability: A verification technique for a class of security kernels. In *Proc.* 5^{th} *Int. Symp. on Programming*, pages 352–362, Berlin, 1982. Springer-Verlag.

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, dec 1992.

[Sch86] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[Smi01] G. Smith. A new type system for secure information flow. In *CSFW14*, pages 115–125. IEEE Computer Society Press, June 2001.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, January 1998. ACM.

[VS97] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19 – 22, 1992. ACM Press.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA, 1980. The Lisp Company.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the ucla unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[WMGT00] P. White, W. Martin, A. Goldberg, and F.S. Taylor. Formal construction of the mathematically analyzed separation kernel. In *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 133–141, September 2000.

[ZL97] Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *Proceedings of the 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

How to prevent type-flaw guessing attacks on password protocols[*][†]

Sreekanth Malladi Jim Alves-Foss

Center for Secure and Dependable Systems
University of Idaho
Moscow, ID 83843, USA
{msskanth,jimaf}@cs.uidaho.edu

Abstract

A message in a protocol is said to have a *type-flaw* if it was created with some intended type, but is later received and treated as a different type. A *type-flaw guessing attack* is an attack where a password is guessed and verified by inducing type-flaws in a protocol.

Heather et al. [HLS00] prove that attacks that use type-flaws can be prevented if honest agents tag messages with their intended types. However, their tagging scheme cannot be used in a password protocol since it allows a guess to be directly verified using the tags inside password encryptions.

In this paper we prove that, by following a modification of Heather et al.'s scheme, most type-flaw guessing attacks can still be prevented.

1 Introduction

Numerous protocols have been introduced to initialize security services for protocol users. One of the goals of these protocols is authentication of a sender's identity. There exists a class of protocols called password protocols, that use user chosen passwords for authentication. If these protocols are not designed well, they may be subject to *guessing attacks* [GLNS93]; here an attacker can learn the password by guessing it and verifying the guess using the messages in the protocol.

A message in a protocol is said to have a *type-flaw* if it was created with an intended type but is later received and treated as a different type. For example, receiving a nonce and treating it as though it was an agent's identity. A *type-flaw guessing attack* is an attack where a type flaw is induced in a protocol to enable a password guessing attack (see appendix for an example).

Heather et al. in [HLS00] proved that attacks involving type-flaws can be prevented if all messages are tagged with their types. For example, in their scheme, a nonce na should be tagged as (nonce, na), an agent's identity a as, (agent, a) and so on.

However, there is a problem with Heather et al.'s solution. Consider the message:

$$\{na\}_{passwd(a,b)}{}^{1}$$

An attacker can attempt a guessing attack by guessing the password $passwd(a, b)$. For example, if the user name is "Arnold Schwarzenegger", "`terminator`" wouldn't be a bad guess for $passwd(a, b)$. If the attacker knows na, he can decrypt $\{na\}_{passwd(a,b)}$ with "`terminator`" to see if it matches the na he knows. If so, that verifies the guess. Otherwise, he can try using another guess.

Note that this attack is not feasible if the attacker does not know na initially. But consider the same message using Heather et al.'s scheme of type-tagging:

[*] We dedicate this paper to Late Professor Roger Needham (1935-2003).

[†] This work was funded by DARPA under grant F30602-02-1-0178.

[1] Here na is a nonce; $\{na\}_{passwd(a,b)}$ represents na encrypted with $passwd(a, b)$.

$$\{\mathsf{nonce}, na\}_{passwd(a,b)}$$

The attacker can decrypt with the guess and see if there is the tag "nonce" in it. If so, that would directly verify the guess. He doesn't even need to know na! Therefore, Heather et al.'s solution against type-flaw attacks cannot be used in password protocols.

We have run into a classic security problem: a security "solution", introduces a new problem.

In this paper, we address this problem by modifying the tagging scheme. We prove that if we follow Heather et al.'s scheme but avoid type-tags inside terms encrypted with passwords, most of the type-flaw guessing attacks can still be prevented.

The only type-flaw that our modified scheme fails to prevent is the following: a password encrypted term, say $\{m1\}_{passwd(a,b)}$ being received, expecting to be of the form $\{m2\}_{passwd(a,b)}$ with $m1$ and $m2$ having different types (ideally). We will have more to say about this case in the Conclusion.

2 Proof Strategy

We introduce a modified version of Heather et al.'s tagging scheme that prevents most type-flaw guessing attacks and does not add redundancy that enables normal guessing attacks. We prove this claim following a model and proof structure very similar to Heather et al. [HLS00].

Our main aim is to prove the following:

> *Whenever there is a guessing attack on a protocol using our tagging scheme, there is an equivalent guessing attack when there are no type-flaws in the protocol.*

Therefore, we prove that:

> *Whenever there is a guessing attack on a protocol using our tagging scheme, there is an equivalent guessing attack when all fields are correctly tagged.*

An off-line guessing attack is characterized by two factors:

- The protocol run (an attacker can actively participate in the protocol run, inducing type-flaws, but doesn't use a guess);
- Attacker inferences from the set of messages in the protocol run that enable him to verify a guess.

Therefore, in order to prove our main claim, we need to prove two things:

1. If an attacker participates in a protocol run C that uses our tagging scheme, then an equivalent protocol run C'' can be visualized in which, every field is correctly tagged;
2. If the attacker can verify a guess from the set of messages in C, then he can also verify a guess from C''.

We use the main result from [HLS00] for point 1 above. Our only modification to their model is the following: We consider all weak encryptions (terms encrypted with passwords) as if they were just another type of atomic elements such as nonce, agent etc. We introduce the modified protocol model and show that the main result in [HLS00] still holds. This is covered in section 3.

For point 2 above, we use the definition for guessing attacks from [CMAE03] and show that, whenever a guess is verifiable from C, then it is also verifiable from C''. This is covered in section 4.

3 Proof Part 1: Heather et al.'s Protocol Model and Main Result

In this section we reiterate the model and main results of Heather et al.'s [HLS00] tagging scheme in the context of our modification.

3.1 Message Structure

Tags, Facts and Taggedfacts

The main message element is a *taggedfact*. It is a combination of a *tag* and *fact*, written as $(tag, fact)$. The idea is that the tag represents the "type" of the fact.

$$TaggedFact ::= Tag \times Fact.$$

Message structures are divided into atoms, pairs and encryptions. An atom is an indivisible element. Sets of atoms are grouped together as $Agent,\ Nonce,\ Pubkey$ and so on. The tags for elements of the these sets are given obvious names such as agent, nonce etc. In our modification we add the set $Wenc$ to $Atoms$ to represent "weak encryptions". The corresponding tag is "wenc", treating weak encryptions as an "abstract type". We will talk more about this set as we progress in the paper.

A pair tag is associated with concatenation of two tagged facts. The tag enc is associated with encryptions together with the collection of tags for the elements inside the encryption and a tag for the key.

$$\begin{aligned} Tag &::= \mathsf{agent} \mid \mathsf{nonce} \mid \mathsf{wenc} \mid \ \ldots \mid \mathsf{pair} \mid \mathsf{enc}\, Tag^*\, Tag \\ Fact &::= Atom \mid \mathtt{PAIR}\, TaggedFact\, TaggedFact \mid \mathtt{ENCRYPT}\, Tag\, TaggedFact\, Fact \end{aligned}$$

An atomic fact a of type "agent", associated with the corresponding tag agent is written as (agent, a). The pairing, $\mathtt{PAIR}\ tf_1\ tf_2$ is written as (tf_1, tf_2). When this is associated with its corresponding tag, "pair", this is written as $(\mathtt{pair}, (tf_1, tf_2))$. $\mathtt{PAIR}\ \mathtt{PAIR}\ tf_1\ tf_2\ tf_3$ should actually be $((tf_1, tf_2), tf_3)$; but it is simply written as (tf_1, tf_2, tf_3) in order to avoid notational clutter, since it is unambigous. A tagged fact tf encrypted with a key k using an algorithm kt is written as $\{tf\}_k^{kt}$. A tag for an encryption, going by the grammar, would look like $\mathsf{enc}\ < t_1, t_2, \ldots, t_n >\ kt$ where $t_1, t_2, \ldots t_n$ are the collection of tags for the facts inside the encryption and kt is the tag for the key. This tag is written in a simpler notation as $\{|t_1, t_2, \ldots, t_n|\}_{kt}$. It is assumed that the tag for the key contains enough information regarding the type of the key (public-key or shared-key etc.) and the encryption algorithm used (RSA, DES etc.)

We extend this message structure by defining the structure of atoms of type wenc as below:

$$\begin{aligned} SubWenc &::= Atom \mid \mathtt{PAIR}\, Subwenc\, Subwenc \mid \mathtt{ENCRYPT}\, Tag\, TaggedFact\, Fact \\ Wenc &::= \mathtt{ENCRYPT}\, Subwenc\, WeakKey \end{aligned}$$

By defining such a structure, we imply that no facts inside a weak encryption is associated with a tag. We will call the set of all such facts as *Subwenc*. We assume that honest agents follow such a structure before encrypting with a weak key (fairly realistic since otherwise, as explained before, the tags themselves would verify a guess).

We split keys into sets called *Strongkeys* and *Weakkeys*, depending on the application of the function to generate the keys. For example application of *Passwd* gives a weak key. In contrast a function *PublicKey* gives rise to a strong key. We will talk more about function applications in section 3.2. We denote a fact $f \in Subwenc$, encrypted with a weakkey $w \in Weakkeys$ as $|\{f\}_w|$.

Projections are defined on tagged facts as:

$$(t, f)_1 \hat{=} t, (t, f)_2 \hat{=} f.$$

A version of the perfect encryption assumption is assumed for strong encryptions, whereby honest agents are capable of knowing if they decrypted an encryption correctly [MCJ97] (called "explicit redundacny" in [Gon90]).

We do not assume any sort of explicit redundancy inside weak encryptions. Usually, the required redundancies for those encryptions, is provided by the protocol itself (called "implicit redundancy" [Gon90]).

Subtaggedfacts

In the following definition, we introduce the subfact relation denoted by '$\sqsubset$' to refer to *subtaggedfacts* of a tagged fact.

Definition 3.1 *The* subfact *relation is the smallest relation on tagged facts such that:*

1. $tf \sqsubset tf$;

2. $tf \sqsubset (t, (tf_1, tf_2))$ *iff* $tf \sqsubset tf_1 \vee tf \sqsubset tf_2$;

3. $tf \sqsubset (t, \{tf'\}_k)$ *iff* $tf \sqsubset tf'$.

Such a relation is also lifted to refer to sub-untagged-facts of a tagged fact. i.e. $f \sqsubset tf$ if $(t, f) \sqsubset tf$ for some tag t.

Correct Tagging

A tagged fact is said to be correctly tagged if it's tag represents the true type of the associated fact. A function "well-tagged" is defined inductively over the structure of tags to represent correct tagging:

$$
\begin{aligned}
\text{well-tagged}(\mathsf{agent}, x) &\Leftrightarrow x \in Agent,\\
\text{well-tagged}(\mathsf{nonce}, x) &\Leftrightarrow x \in Nonce,\\
\text{well-tagged}(\mathsf{wenc}, x) &\Leftrightarrow x \in Wenc,\\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
\text{well-tagged}(\mathsf{pair}, x) \quad \Leftrightarrow \quad &\exists\, tf_1,\\
&tf_2 : TaggedFact \bullet x = \mathtt{PAIR}\, tf_1\, tf_2 \wedge\\
&\text{well-tagged}\, tf_1 \wedge \text{well-tagged}\, tf_2,
\end{aligned}
$$

$$
\begin{aligned}
\text{well-tagged}(\{|ts|\}_{kt}, x) \Leftrightarrow \quad &\exists\, tf : TaggedFact;\\
&k : Fact \bullet x = \{tf\}_k^{kt} \wedge \text{well-tagged}(tf)\\
&\wedge \text{well-tagged}(kt, k) \wedge ts = \text{get-tags}\, tf.
\end{aligned}
$$

where get-tags returns the collective sequence of tags inside an encryption, defined as:

$$
\begin{aligned}
\text{get-tags}(\mathsf{pair}, (tf_1, tf_2)) &= \text{get-tags}\, tf_1 ^\frown \text{get-tags}\, tf_2,\\
\text{get-tags}(t, f) &= \langle\, t\, \rangle, \quad \text{for } t \neq \mathsf{pair}.
\end{aligned}
$$

A well-tagged fact represents a taggedfact which is correctly tagged and has every subtaggedfact in it, correctly tagged. In contrast, a fact is characterized as top-level-well-tagged when a fact is correctly tagged at the outer-most level. This means, for example, a taggedfact is indeed a pair of tagged facts when it's tag equals pair, even if the two tagged facts may not be well-tagged.

$$
\begin{aligned}
\text{top-level-well-tagged}(\mathsf{agent}, x) &\Leftrightarrow x \in Agent,\\
\text{top-level-well-tagged}(\mathsf{nonce}, x) &\Leftrightarrow x \in Nonce,\\
\text{top-level-well-tagged}(\mathsf{wenc}, x) &\Leftrightarrow x \in Wenc,\\
&\ldots
\end{aligned}
$$

$$
\text{top-level-well-tagged}(\mathsf{pair}, x) \Leftrightarrow \exists\, tf_1,\ tf_2 : TaggedFact \bullet x = \mathtt{PAIR}\, tf_1\, tf_2,
$$

$$
\text{top-level-well-tagged}(\{|ts|\}_{kt}, x) \Leftrightarrow \exists\, tf : TaggedFact; k : Fact \bullet x = \{tf\}_k^{kt} \wedge ts = \text{get-tags}\, tf.
$$

3.2 The framework

In the previous section, the structure of messages in a protocol and their properties were introduced. In this section, we introduce the framework on which messages are used to build protocol runs.

The framework is derived from the strand space model of [THG99]. A *strand* is a sequence of communications represented as $< \pm tf_1, \pm tf_2, \ldots, \pm tf_n >$. $+tf$ indicates sending tf and $-tf$ indicates receiving tf. Each send or receive event is a *node*. A transition from consecutive nodes n_i and n_{i+1} on the same strand is represented as $n_i \Rightarrow n_{i+1}$. A transmission of a tagged fact from n_i on one strand, followed by a reception in n_j on another strand is represented as $n_i \rightarrow n_j$.

A *bundle* represents a partial or complete protocol run. It is an acyclic digraph using edges $\rightarrow$ and $\Rightarrow$ such that, whenever a tagged fact is received, the bundle also includes a transmission of the tagged fact. Further, a bundle holds the history of the network from the starting of the communication.

A node is said to be an *entry point* to a set of tagged facts if no previous node has uttered an element of that set. A taggedfact is said to be *originating* on a node if the node is an entry point for the set to which the taggedfact belongs. A taggedfact is said to be *uniquely originating* if there is no other node in the bundle that utters an element of the set to which the tagged fact belongs.

3.3 Honest strands

Honest strands represent execution traces of honest agents. Since roles of honest agents is dictated by the protocol (in terms of sending and receiving messages), it makes sense to have some set of "templates" that dictate the actions of those roles in the protocol. Therefore strand templates are defined which specify the message structure of honest agents under ideal conditions. These contain variables that would be instantiated to output honest strands.

Each taggedfact in an honest strand corresponds to an instantiation of a "tagged template" in a strand template. Tagged templates are defined by the following grammar:

$$\begin{array}{rcl} TaggedTemplate & ::= & Tag \times \; Template \\ Template & ::= & Var \mid \texttt{APPLY}\, F_n \, Var^* \mid \texttt{PAIR}\, TaggedTemplate\, TaggedTemplate \mid \\ & & \texttt{ENCRYPT}\, Tag\, TaggedTemplate\, TaggedTemplate \end{array}$$

Here Var represents atomic variables, which upon instantiation output atomic facts. $\texttt{APPLY}\ F_n\ Var^*$ means that a function identifier F_n is being applied to a collection of atomic variables. This is application is the basis to generate keys, hashes of messages etc. For example, in $PublicKey(A), F_n = PublicKey$. Note that this specification allows to model constructed keys, not just atomic keys, which is important for 'real-world' protocols such as `SSL 3.0`. (Atomic keys refer to the keys possessed by partipants which are handled by exhaustive substitution of agents' identities. Constructed keys are keys produced from just about any random bitstring formed using different message elements).

The next step is to consider how tagged templates are instantiated to form taggedfacts. This is accomplished by defining an instantiation function sub to substitute facts for variables:

$$sub \,:\, Var \rightarrow Fact$$

The properties of this function are defined below in order to instantiate all possible tagged templates:

$$\begin{array}{rcl} sub(t, v) & = & (t, sub(v)) \quad \text{for } v \in Var, \\ sub(t, g(v_1, \ldots, v_n)) & = & (t, g(sub(v_1), \ldots, sub(v_n))), \\ & & \text{where } g \in Fn, \text{ and } Fn \text{ is the} \\ & & \text{set of function identifiers.} \\ sub(\mathsf{pair}, (tt_1, tt_2)) & = & (\mathsf{pair}, (sub(tt_1), sub(tt_2))), \\ sub(\{|ts|\}_{tk}, \{tt\}_k) & = & (\{|ts|\}_{tk}, \{sub(tt)\}_{sub(tk,k)_2}), \\ & & \text{where } k = g(v_1, \ldots, v_n) \text{and } g \in Fn \text{ represents a key} \\ & & \text{type using a particular keying algorithm.} \end{array}$$

For the third and fourth clauses above, there is a little change from the same expressions given in [HLS00]. (They use tf_1, tf_2 and tf in place of tt_1, tt_2 and tt. However, since sub is an instantiation of variables and not facts, we feel it is proper to apply it on templates instead of facts. This change however, wouldn't affect their results in any way).

There are two assumptions on strand templates and instantiating templates:

1. For every strand template, there is some ideal tag environment ρ defined as:

   $$\rho : (Var \rightarrow Tag) \cup (Fn \rightarrow Tag^* \times Tag)$$

   The idea is that ρ returns the tags for each variable in a template. This is to ensure that the same tags are always given to the same variables in a template. (For the exact properties of ρ, please refer [HLS00].)

2. If a taggedfact tf originates on a honest strand, then top-level-well-tagged(tf).

   This means, it is assumed that honest agents always tag messages correctly. However, since it is impossible to distinguish between random bitstrings, it is probably more appropriate to say, whenever a bitstring is substituted for a variable next to a tag in a template, then the bitstring is automatically added to the set corresponding to that tag. (For example instantiating N_A in (nonce, na) would result in N_A being added to the set $Nonce$.) The bitstring is treated to be of that type from then onwards.

3.4 Penetrator strands

The penetrator is considered to have standard Dolev-Yao attacker capabilities. i.e. She can overhear messages on a network, construct messages, split them, send her own messages and so on. She is also assumed to possess some set K_P of keys and prodcue some texts T of her choice. These capabilities are listed in the following definition.

Definition 3.2 *An* on-line penetrator strand *is one of the following:*

M	*Text message*	$\langle +(t,x)\rangle$ *with well-tagged*(t,x) *and* $x \in T$.
F	*flushing*	$\langle -tf\rangle$.
T	*Tee*	$\langle -tf, +tf, +tf\rangle$.
C	*Concatenation*	$\langle -tf, -tf', +(\mathsf{pair}, (tf, tf'))\rangle$.
S	*Separation*	$\langle -(\mathsf{pair}, (tf, tf')), +tf, +tf'\rangle$.
K	*Key*	$\langle +(tk,k)\rangle$ *with well-tagged*(tk,k) *and* $k \in K_P$.
E	*Encryption*	$\langle -(tk,k), -tf, +(\{\lvert ts\rvert\}_{tk}, \{tf\}_k^{tk})\rangle$, *where* $ts =$ *get-tags*(tf).
D	*Decryption*	$\langle -(tk',k'), -(\{\lvert t\rvert\}_{tk}, \{tf\}_k^{tk}), +tf\rangle$, *where* tk *and* tk' *are tags representing inverse key types, and* k' *is the corresponding decrypting key of* k *with both being of the type* tk *and* tk' *respectively.*
R	*Retagging*	$\langle -(t,f), +(t',f)\rangle$.

The retagging strand captures the concept of receiving a message of one type and sending it, with a claim of a different type. In this paper, we will denote the set of on-line penetrator strands as $\mathbf{X_{on}}$. In Section 4 we will add some more strands to the above capabilities to model off-line guessing attacks.

Note that, we treat weak encryptions as an "abstract type". i.e. we do not allow the attacker to perform any operations on it during the on-line communication. We also assume that guessing the password and deducing the contents inside the encryption is done entirely off-line. Lastly, we consider only those attacks in which the attacker is able to learn a password shared by honest agents by attempting an off-line guessing attack. In other words, we do not consider attacks wherein a password is learned by breaching secrecy.

3.5 Transforming arbitrarily tagged bundles to well-tagged bundles

An arbitrarily tagged bundle represents a bundle with or without type-flaws. Since a tag in a taggedfact indicates the type of it's fact, a correctly tagged fact indicates that the fact is *indeed* the type indicated by it's tag. Generally speaking, a well-tagged bundle represents that all it's tagged facts are correctly tagged. This in turn means that there are no type-flaws in a well-tagged bundle. The main result in [HLS00] states that any bundle that uses the tagging scheme can be changed into an equivalent well-tagged bundle.

To prove this hypothesis, Heather et al. define a *renaming function* that changes any arbitrarily tagged bundle to a well-tagged bundle. The main idea behind such a transformation being possible is that, if an honest agent is willing to accept an ill-tagged fact (t, f), then it should accept any value in place of f. Naturally, this includes the fact f' such that well-tagged(t, f').[2]

Below is the definition and properties of the renaming transformation:

Definition 3.3

$$\phi : TaggedFact \rightarrow TaggedFact$$

is a renaming function *having the following properties:*

[2]There seems to be a typo in [HLS00] in stating the same.

1. *ϕ preserves top-level tags:*

$$\phi(t, f) = (t', f') \Rightarrow t = t';$$

2. *ϕ returns well-tagged terms: well-tagged($\phi(tf)$);*

3. *ϕ is the identity function over well-tagged terms:*

$$\textit{well-tagged}(tf) \Rightarrow \phi(tf) = tf;$$

4. *ϕ distributes through concatenations that are top-level-well-tagged:*

$$\phi(tf_1, tf_2) = (\phi(tf_1), \phi(tf_2));$$

5. *ϕ distributes through encryptions that are top-level-well-tagged:*

$$\phi(\{|ts|\}_{kt}, \{tf\}_k^{tk}) = \quad (\{|ts|\}_{kt}, \{\phi(tf)\}_{\phi(tk,k)_2}^{tk}) \textit{ if } ts = \textit{get-tags}(tf);$$

6. *ϕ respects inverses of keys: if (tk, k) and (tk', k') are inverses of each other, then so are $\phi(tk, k)$ and $\phi(tk', k')$, tk and tk' being their types;*

7. *When ϕ is applied to a top-level-ill-tagged fact (t, f) of C, such that $\phi(t, f) = (t, f')$, then $f' \in T$;*

8. *When ϕ is applied to a top-level-ill-tagged fact tf of C, it produces a fact that has an essentially new value. i.e., a fact that has no sub-untagged-fact in common with $\phi(tf')_2$ for any other fact tf' of C:*

$$\forall tf \in facts(C) \,.\, \neg\textit{top-level-well-tagged}(tf) \wedge f \sqsubset \phi(tf) \Rightarrow \forall tf' \in facts(C) \;.\; tf \not\sqsubset tf' \Rightarrow f \not\sqsubset \phi(tf').$$

where $facts(C)$ represents all the facts and sub-untagged-facts of nodes in C.

This establishes an injectivity property for ϕ over facts of C.

Merely defining such a renaming transformation neither proves that all possible taggedfacts in C are covered by ϕ nor proves that the $\phi(C)$ is a bundle by definition. Therefore, there are essentially four things to be proven:

1. Given a bundle C, there is some renaming function ϕ for C. (Refer [HLS00, Lemma 3]).

2. If $temp$ is a template for an honest agent and $sub(temp)$ is an instantiation of the template, then $\phi(sub(temp))$ corresponds to an instantiation of the same template using some other function sub'. i.e.

$$\phi(sub(temp)) = sub'(temp).$$

   This means if $sub(temp)$ is an honest strand, then $sub'(temp)$ is also an honest strand. (Refer [HLS00, Lemma 4]).

3. The penetrator is "equally capable" in C and $\phi(C)$. In other words, if X is a penetrator strand in C, then X is also a penetrator strand in $\phi(C)$ with every tagged tf in X replaced by $\phi(tf)$. (Refer [HLS00, Section 3.3].)

4. Protocol security is entirely based on values that originate uniquely, such as nonces and short term keys. Therefore, it is important to ensure that the transformed bundle doesn't contain nodes that "duplicate" such values. To this end, a bundle C'' is produced from $\phi(C)$ such that, facts in C'' are uniquely originating if they were uniquely originating in C. (Refer [HLS00, section 3.4].)

Since our modification only defines a new subset of the atoms, the proofs for the above points presented in Heather et al. [HLS00] would still hold.

3.6 Main Result

The main result of Heather et al. ([HLS00, Theorem 1]) follows from the concepts explained in the previous section:

Theorem 3.4 *If C is a bundle (under the tagging scheme) then there is a renaming function ϕ and a bundle C'', such that:*

- *C'' contains the tagged facts of C (considered as a set), renamed by ϕ;*
- *C'' contains the same honest strands as C, modulo some renaming;*
- *facts are uniquely originating in C'' if they were uniquely originating in C;*
- *all tagged facts in C'' are well-tagged.*

4 Proof part 2 : Off-line guessing attacks

4.1 Off-line penetrator capabilities

In this section we will introduce our notion of an attacker engaging in off-line guessing and verification. We assume a set G of guesses that a penetrator possesses. In the off-line phase, a penetrator can guess a password, use it to encrypt and decrypt weak encryptions, concatenate and split facts, tag facts and untag tagged facts. We add some more penetrator strands to the capabilities in definition 3.2 to capture such off-line capabilities:

$\mathbf{D_g}$	$Decryption\_using\_Guess$ $\langle -\lvert\{f\}_g\rvert, -g, +f\rangle$ with $g \in G$.
$\mathbf{E_g}$	$Encryption\_using\_Guess$ $\langle -f, -g, +\lvert\{f\}_g\rvert\rangle$ with $g \in G$.
$\mathbf{C_f}$	$Concatenating\_facts$ $\langle -f, -f', +(f, f')\rangle$.
$\mathbf{S_f}$	$Separating\_facts$ $\langle -(f, f'), +f, +f'\rangle$.
$\mathbf{Tg}$	$Tagging$ $\langle -t, -f, +(t, f)\rangle$.
$\mathbf{Utg}$	$Untagging$ $\langle -(t, f), +f\rangle$.

Note that in the off-line phase, the penetrator can still use some of his on-line capabilities. However, since there wouldn't be any network events during the off-line phase, we will remove some strands from $\mathbf{X_{on}}$ before adding them to off-line capabilities:

Definition 4.1 *The set of* off-line penetrator strands*, denoted as $\mathbf{X_{off}}$ is defined as:*

$$\mathbf{X_{off}} = \mathbf{X_{on}} \setminus \{\mathbf{M}, \mathbf{F}, \mathbf{T}, \mathbf{K}, \mathbf{R}\} \cup \{\mathbf{D_g}, \mathbf{E_g}, \mathbf{C_f}, \mathbf{S_f}, \mathbf{Tg}, \mathbf{Utg}\}$$
or
$$\mathbf{X_{off}} = \{\mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{D}, \mathbf{D_g}, \mathbf{E_g}, \mathbf{C_f}, \mathbf{S_f}, \mathbf{Tg}, \mathbf{Utg}\}.$$

4.2 Defining off-line guessing attacks

Before giving a formal definition for guessing attacks, we define a relation *deducible* such that, tf is deducible from a bundle C, if there is a valid sequence of penetrator strands that yield tf from C.

Firstly, we introduce a simple inference relation $\vdash$. If S a set of tagged facts, we write $S \vdash_X tf$ if the strand X can be constructed such that, for every $tf' \in S$, tf' occurs on a '$-$' node in X, and tf is a tagged fact on any '$+$' node of X.

Definition 4.2 *Let C be a bundle. Then, tf_n is* deducible *from C, or:*

$$C \models_{tr} tf_n$$

if $tr = < S_1 \vdash_{X_1} tf_1, S_2 \vdash_{X_2} tf_2, \ldots, S_n \vdash_{X_n} tf_n >$ such that, for $i = 1 \ldots n, X_i \in \mathbf{X_{off}} \setminus \{\mathbf{D_g}, \mathbf{E_g}\}$ and $S_{i+1} \subseteq Taggedfacts(C) \cup \{tf_1, \ldots, tf_i\}$, where $Taggedfacts(C)$ is the set of taggedfacts on all the nodes in C.

We will tend to drop the subscript tr when it is obvious. Note that guesses will not be present in C, and have to be added to construct $\mathbf{D_g}$ and $\mathbf{E_g}$ strands from C.

Lemma 4.3 *Let C and $C^{''}$ be two bundles defined as in section 3. Then,*

$$C \cup \{g\} \models_{tr} tf \Rightarrow C^{''} \cup \{g\} \models_{\phi(tr)} \phi(tf).$$

Proof: We need to show that, for every possible inference $S \vdash_X tf$ in tr, there is an equivalent $\phi(S) \vdash_{\phi(X)} \phi(tf)$ in $\phi(tr)$. This inturn implies we need to show that for every possible strand $X \in \mathbf{X_{off}} \cup \{\mathbf{D_g}, \mathbf{E_g}\}$ from C, there is an equivalent strand from $C^{''}$ such that, $\phi(X) \in \phi(\mathbf{X_{off}} \cup \{\mathbf{D_g}, \mathbf{E_g}\})$.

It is proven in [HLS00, section 3.3] that for each of the penetrator strands in C, equivalent penetrator strands in $C^{''}$ can be constructed. Therefore, for every $X \in \{\mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{D}\}$, there is a corresponding $\phi(X) \in \{\phi(\mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{D})\}$.

We now prove that for every $X \in \{\mathbf{D_g}, \mathbf{E_g}, \mathbf{C_f}, \mathbf{S_f}, \mathbf{Tg}, \mathbf{Utg}\}$ there is a corresponding $X' = \phi(X)$ such that, $X' \in \{\phi(\mathbf{D_g}, \mathbf{E_g}, \mathbf{C_f}, \mathbf{S_f}, \mathbf{Tg}, \mathbf{Utg})\}$.

- If X is a $\mathbf{D_g}$ strand, then

$$X' = \langle -\phi(\mathsf{wenc}, |\{f\}_g|)_2, -g, +f \rangle,$$

which is a $\mathbf{D_g}$ strand because, when $(\mathsf{wenc}, |\{f\}_g|)$ originates on a regular node in C, then well-tagged$(\mathsf{wenc}, |\{f\}_g|)$ and therefore, $\phi(\mathsf{wenc}, |\{f\}_g|) = (\mathsf{wenc}, |\{f\}_g|)$.

- If X is a $\mathbf{E_G}$ strand, then

$$X' = \langle -\phi(t, f)_2, -g, +|\{\phi(t, f)_2\}_g| \rangle,$$

which is an $\mathbf{E_G}$ strand. (Note that $\phi(t, f)$ is not defined for all $f \in Subwenc$, in which case, we replace $\phi(t, f)_2$ with f itself.)

- If X is a $\mathbf{C_f}$ strand, then

$$X' = \langle -\phi(t, f)_2, -\phi(t, f')_2, +(\phi(t, f)_2, \phi(t, f')_2) \rangle,$$

which is a $\mathbf{C_f}$ strand.

- If X is a $\mathbf{S_f}$ strand, then

$$X' = \langle +(\phi(t, f)_2, \phi(t, f')_2), +\phi(t, f)_2, +\phi(t, f')_2, \rangle,$$

which is a $\mathbf{S_f}$ strand.

- If X is a $\mathbf{Tg}$ strand, then

$$X' = \langle -t, -\phi(t, f)_2, +\phi(t, f) \rangle.$$

Now $\phi(t, f) = (t, f')$ for some f' such that well-tagged(t, f'). Therefore, we can rewrite the above expression as $X' = \langle -t, -f', +(t, f') \rangle$, which is a $\mathbf{Tg}$ strand.

- If X is a $\mathbf{Utg}$ strand, then

$$X' = \langle -\phi(t, f), +\phi(t, f)_2 \rangle.$$

Again, since $\phi(t, f)_2 = (t, f')$ for some f' such that well-tagged(t, f'), this can be rewritten as $X' = \langle -(t, f'), +f' \rangle$, which is a $\mathbf{Utg}$ strand.

$\square$

Corollary 4.4

$$C \not\models tf \Rightarrow C^{''} \not\models \phi(tf).$$

Now the above expression is equivalent to,

$$C^{''} \models_{\phi(tr)} \phi(tf) \Rightarrow C \models_{tr} tf.$$

Above we proved that every possible inference $\phi(S) \vdash_{\phi(X)} \phi(tf)$ in $\phi(tr)$ from $C^{''}$ corresponds to an equivalent $S \vdash_X tf$ in tr from C. Hence, the result.

We now give a simple definition for a guessing attack. We say that a *guessing attack* is possible on a bundle C, if a guess $g \in G$ is *verifiable* in C.

In short, we try to see if the attacker can deduce a taggedfact in *atmost* one way before guessing, but in more than one way after guessing. To find if there are two different ways to deduce a taggedfact, we 'mask' the first occurence with some random value and then look for another occurence of it.

Definition 4.5 *Let C and g be as defined above. Let sub be an instantiation function for a template $temp$ such that $sub(temp) \in C$ and tt be a tagged template in $temp$; Also let $tf = sub(tt)$. Then, g is* verifiable *from C and tf is a* verifier *for g iff:*

$$\hat{C} \cup \{g\} \models tf \ \wedge \ \hat{C} \cup \{g\} \models \hat{tf}; \text{ and} \tag{1}$$

$$\hat{C} \not\models tf \vee \hat{C} \not\models \hat{tf}. \tag{2}$$

where $\hat{tf}$ is a fresh constant and $\hat{C}$ is obtained by replacing the particular occurrence of tf in C, with $\hat{tf}$.

4.3 The main result

Our main aim is to show that, whenever there is a guessing attack on C, there is also a guessing attack on $C^{''}$. If there is a guessing attack on C, by definition, a guess $g \in G$ is verifiable in C with a verifier $sub(tt)$. Therefore, we frame our main theorem as,

Theorem 4.6 *Whenever $g \in G$ is verifiable from C, g is also verifiable from $C^{''}$.*

Proof:
Let sub' be defined as in section 3.3:

$$sub'(tt) = \phi(sub(tt)).$$

Let $C^{''}$ be denoted as $\mathcal{C}$ and $\phi(tf)$ as tf'. From Lemma 4.3, $C \cup \{g\} \models tf \Rightarrow \mathcal{C} \cup \{g\} \models \phi(tf)$. Further, from Corollary 4.4, $C \not\models tf \Rightarrow \mathcal{C} \not\models \phi(tf)$. Therefore, 1 and 2 in definition 4.5 can be written as,

$$\hat{\mathcal{C}} \cup \{g\} \models tf' \ \wedge \ \hat{\mathcal{C}} \cup \{g\} \models \hat{tf'}; \text{ and} \tag{3}$$

$$\hat{\mathcal{C}} \not\models tf' \vee \hat{\mathcal{C}} \not\models \hat{tf'}. \tag{4}$$

Also, $\phi(tf) = \phi(sub(tt)) = sub'(tt)$. Therefore, g is verifiable in $\mathcal{C}$ with a verifier $sub'(tt)$.

□

5 Conclusion

In this paper we have considered type-flaw guessing attacks on password protocols. We modified Heather et al.'s existing solution to prevent type-flaw attacks and proved that such modification prevents type-flaw guessing attacks on password protocols. Our proof strategy was built on Heather et al's proof structure with a minor change: We considered all weak encryptions as atoms. This was possible since we disallowed any attacker operations on such terms.

Our proof proceeded in two stages:

1. The on-line communication: here we proved that basically the same protocol run is obtained when all messages are correctly tagged, if it was obtained by adopting our tagging scheme. Most of this result was already established by Heather et al. A renaming function is applied on an arbitrarily tagged bundle so that the resulting bundle has every message correctly tagged. Such a renaming is realistic because, if an honest agent is willing to accept an ill-tagged message, it should accept any value in it's place;

2. We showed that a guessing attack is possible on the correctly tagged bundle, if it was possible on the original bundle. This indirectly proves that the attack was not based on a type-flaw but on some other mechanism.

The implementation of the tagging scheme using bit strings can be referred from [HLS00].

In the following section we will discuss some interesting issues together with directions towards future work.

5.1 Discussion and Future work

Observe that our proof (or for that matter Heather et al.'s proof) is highly dependent on the way a type-flaw is defined. i.e. for example, if we define that sending an atom of one type, claiming it as an atom of another type is not a type-flaw, then the tag structure would appear as follows:

$$Tag ::= \mathsf{atom} | \ \mathsf{pair} \ | \ \mathsf{enc} \ Tag^* \ Tag.$$

Such a tagging would allow for example, sending a key, claiming it as an agent's identity but prevents sending an atom as a pair or as a (strong) encryption.

Similarly, we identified all weak encryptions, regardless of their structure, as belonging to a unique type, wenc. Therefore, it would allow weak encryptions having different structures to be replayed in place of one another. For example, a message $\{na, k, nb\}_{passwd(a)}$ can be replayed, claiming it to be structurally identical to $\{k, na, ts\}_{passwd(a)}$ (na, nb are nonces. k is a key and ts is a timestamp). Such type-flaws may be used in attacks but can neither be prevented by our tagging scheme nor our proof establishes that they cannot be used in attacks.

However, in practice, many times such replays can be avoided. For example, consider the following messages in Gong et al.'s popular, "Demonstration protocol" [GLNS93]:

$$\begin{array}{l} \text{Msg 1. } a \to s : \{a, b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pk(s)} \\ \text{Msg 4. } s \to a : \{na1, na2 \oplus k\}_{passwd(a)}. \end{array}$$

Here ca is a redundant random number. $pk(s)$ is the public-key of s. Under some assumptions about message structures, a type-flaw guessing attack is possible on this protocol. An attacker can use Msg 4 in a legitimate run between a and s as follows:

$$\begin{array}{lll} \text{Msg 1. } I(a) \to s & : & \{a, b, nI1, nI2, ca, \{na1, na2 \oplus k\}_{passwd(a)}\}_{pk(s)} \\ \text{Msg 4. } s \to I(a) & : & \{nI1, nI2 \oplus k'\}_{passwd(a)}. \end{array}$$

$I(a)$ denotes attacker I pretending as a. The attacker creates his own nonces $nI1$ and $nI2$ together with Msg 4 of the previous run to construct Msg 1 and sends it to s. After he gets back Msg 4 from s as a response, he decrypts it with a guess and matches the first part ($nI1$) with his $nI1$ to verify the guess.

The other messages of the protocol are irrelevant in this attack.

Now this attack can be prevented if there is a tag for the time stamp ts in Msg 1. This type tag would not directly verify a guess because it is protected by another layer of encryption under a strong key ($pk(s)$).

Some replays cannot be avoided. For example, $\{f\}_{passwd(a)}$ can be replayed in $\{f'\}_{passwd(a)}$ provided f and f' can be "unified". However, in most cases, the possibility of such unification itself means that a guessing attack is possible: since unification implies that constants in f and f' should match, whenever f and f' are textually distinct (except for the positions of the constants), the constants would themselves verify a guess. For example, $\{na, K, NB\}_{passwd(a)}$ can be unified with $\{na, Ts, K\}_{passwd(a)}$ (na is constant, K, NB are variables). However, na can be obtained from both messages in two different ways, by using a guess; this verifies the guess even before unification!

Observe that in the tagging scheme, tags not protected by encryption can be safely removed while acheiving the same results. Further, the tags inside encryptions can be combined into a single *component number*. As Heather et al. argue, this simplication is *fault-preserving* in the sense of Hui and Lowe [HL01]: That means, if there is an attack on the component numbering scheme, there was also an attack on the original tagging scheme.

Such component numbering ensures that encrypted components can not be replayed in place of one another. Above we argued (although yet to prove formally), that weak encryptions should as well be non-replayable (i.e. non-unifiable). Therefore, a protocol following this numbering suggestion, along with the component numbering scheme, ensures that no replays of encrypted components are possible. Such a result in protocol analysis has already been shown in numerous occasions as holding the key to protocol security [AN94, GT00]. Fairly recently, it was also shown to ensure decidability for security protocols in the context of secrecy [RS03]. (Secrecy is a security property that specifies that an attacker should not be able to learn a secret value from a protocol run.)

We also believe that the result regarding component numbering makes it easy to prove that "protocol numbering" inside encrypted components would prevent *multi-protocol guessing attacks* [MAFM02, CMAE03] if we can find a way to enforce the numbering. (A multi-protocol guessing attack works by replaying encrypted components from one protocol into a different protocol.)

Observe that we assume sufficient redundancy inside strong encryptions that allows honest agents to know if they decrypted them correctly. However, we did not allow such a redundancy in weak encryptions because that may verify a guess directly [Gon90]. In contrast, Lowe states that redundacy inside any encryptions (including strong) would aid in guessing attacks [Low02]. However, without the redundancies it is hard to see how honest agents can run protocols, satisfactorily.

Secrecy and guessing attacks seem to be quite more integrally related than what meets the eye. Halevi et al. have shown that security against guessing attacks can be reduced to the initial problem of establishing a secret between two unfamiliar parties [HK99]. (A corollary is that public key encryption is unavoidable to solve both the problems.) Thus, it is not entirely surprising that the same problems and solutions encountered in studying secrecy attacks on protocols also apply for guessing attacks.

Observe that learning a password through a guessing attack can result in breaches of secrecy not known to exist when analysing protocols for secrecy. For example, a successful guessing attack is possible on $\{na, nb\}_{passwd(a)}$ and na, but attacker also learns an otherwise secret nb.

Also observe that, like secrecy and authentication, guessing attacks should also be studied as a *trace property* (a trace property is a security property that can be verified by examining all possible traces or protocol runs within a scenario). Therefore, it would be interesting to see if the same results regarding decidability that were published for secrecy and authentication apply for guessing attacks as well (eg. [MS01, Low99]).

The ideal tag environment ρ defined in section 3 assumes more importance than it may seem. A necessary condition for successful use of the tagging scheme is that *all* honest agents follow the same implementation. For example, agent a cannot run a protocol using value 001 for the tag nonce with b, who uses another value, say 101 for the same tag. This is also true when a itself is involved in different runs of the same protocol or if it is simultaneously engaging in runs from different protocols (eg. `SSL 3.0` and `SET` concurrently). However, Heather et al.'s formal definition of ρ only specifies that each of the honest roles need to have tag values that are consistent within the same template; they do not specify that *all* honest agents follow the same tag values, which we believe is inadequate. Of course, it is also hard to have such "universally-agreed upon" tag values without having some sort of "international standards" for tagging schemes. And, there is no guarantee that malicious code will not use the wrong tag values to deliberately tailor a protocol to use for attacks [AF98].

In this paper we have considered the definition for guessing attacks given in [CMAE03] which only considers verifiers that are subterms of the attacker's initial knowledge. This definition is specifically tailored to the standard inference rules. In contrast, Lowe's definition in [Low02] is stronger in this sense, because it can be used for any attacker inference set. (For example the rule $\{m, n\}_k \vdash \{m\}_k$ is not in the standard inference set, but holds when using Cipher Block Chaining.) It would be interesting to see how this affects the results in this paper.

However, regardless of how such inference rules affect the results, they can be used in attacking Heather et al.'s original scheme as well (See Appendix for an attack on the Woo and Lam authentication protocol $\pi 1$ [WL94]).

There are two other unsolved issues in Heather et al.'s scheme:

1. They do not consider all possible forms of constructed keys (but only those that result from application of a key function F_n to concatenation of sequence of atoms $(f_1, \ldots, f_n)$);

2. They do not consider cancellativity and other algebraic properties obeyed by message elements when using operations such as products and `XOR` [MS03]. (these operations are frequently used in real-world protocols).

Lastly, we did not consider implementation dependent guessing attacks in this paper. For example, the password can be learned from $\{english\_text\}_{passwd(a)}$ by decrypting it with a guess (even though $english\_text$ is not known initially).

We look forward to the future with all the issues pointed out in this section, which will keep us busy.

Acknowledgments

We would like to thank Iliano Cervesato and the anonymous referrees for insightful comments. Thanks are also due to Ricardo Corin for many helpful technical discussions.

Bibliography

[AN94] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6-15, 1996.

[AF98] J. Alves-Foss. Multi-Protocol Attacks and the Public-Key Infrastructure. *21st National Information Systems Security Conference*, 566-576, October 1998.

[CMAE03] R. Corin, S. Malladi, J. Alves-Foss and S. Etalle. Guess what? Here is a new tool that finds some new guessing attacks. *Workshop on Issues in the Theory of Security (WITS'03)*, Warsaw, March 2003.

[Gon90] L. Gong. A note on redundancy in encrypted messages, *ACM Computer Communication Review*, 20(5), 18-22, 1990.

[GLNS93] L. Gong, T. M. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.

[GT00] J. D. Guttman and F. J. Thayer. Protocol independence through disjoint encryption. *13th IEEE Computer Security Foundations Workshop*, 24–34, July 2000.

[HK99] S. Halevi and H. Krawczyk. Public-Key cryptography and password protocols. *ACM Transactions on Information and System Security*, 2(3): 230-268, 1999.

[HLS00] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings, 13th Computer Security Foundations Workshop*, 255-268, , July 2000.

[HL01] M. L. Hui and G. Lowe. Fault-preserving safe simplifying transformations on security protocols, *Journal of Computer Security*, volume 9, 3-46, 2001.

[Low99] G. Lowe. Towards a completeness result for model-checking of security protocols, *Journal of Computer Security*, volume (7), 89-146, 1999.

[Low02] G. Lowe. Analysing protocols subject to guessing attacks. *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, January 2002.

[MAFM02] S. Malladi, J. Alves-Foss, and S. Malladi. What are multi-protocol guessing attacks and how to prevent them. *Eleventh IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE-Enterprise Security*, 77-82, June 2002.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols, 1997.* Available via URL: `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html.`

[MS01] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. *ACM Conference on Computer and Communication Security*, 166–175, 2001.

[MS03] J. Millen and V. Shmatikov. Symbolic protocol analysis with products and diffie-hellman key exponentiation. *16th IEEE Computer Security Foundations Workshop*, July 2003.

[RS03] R. Ramanujam and S. P. Suresh. A decidable subclass of unbounded security protocols. *Workshop on Issues in the Theory of Security (WITS'02)*, Warsaw, March 2003.

[THG99] F. J. Thayer, J.C. Herzog, and J.D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*,7(2,3):191-230, 1999.

[WL94] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. *Operating Systems Review,*,28(3):24-37, 1994.

Appendix A : Example for type-flaw guessing attack

Lomas et al. presented the following protocol in [GLNS93] (call it $\mathbf{P1}$):

$$\text{Msg 1. } A \to B : \{C, N\}_{pk(B)}$$
$$\text{Msg 2. } B \to A : \{f(N)\}_{PAB}.$$

C is a redundant random number called "confounder". N is an integer value and f is an easily invertible function. PAB is $passwd(A, B)$.

A type-flaw guessing attack is possible on $\mathbf{P1}$.

Let $\mathbf{P1}$ be implemented so that the first message is $\{N, C\}_{pk(B)}$ instead of $\{C, N\}_{pk(B)}$. This apparently inconsequential change leads to an attack.

Msg 1. $A \rightarrow B : \{N, C\}_{pk(B)}$
Msg 2. $B \rightarrow A : \{f(N)\}_{PAB}$.

Call this $\mathbf{P2}$. When $\mathbf{P1}$ and $\mathbf{P2}$ are combined, the scenario looks like:

Msg P1.1. $a \rightarrow b : \{c, n\}_{pk(b)}$
Msg P1.2. $b \rightarrow a : \{f(n)\}_{pab}$
Msg P2.1. $I(a) \rightarrow b : \{c, n\}_{pk(b)}$
Msg P2.2. $b \rightarrow I(a) : \{f(c)\}_{pab}$.

An off-line guessing attack on this protocol run is reported in the following trace:

```
guesses:[pab], verifier: [c,n] * pk(b)
verification trace:  sdec(c + pab), sdec(n + pab), keylookup(pk(b)),
pair([c,n]), penc([c,n] * pk(b)).
```

Attacker can replay msg 1 of $\mathbf{P1}$ in $\mathbf{P2}$. After b sends msg 2 in $\mathbf{P2}$, attacker now has $\{f(n)\}_{pab}$ and $\{f(c)\}_{pab}$. She can decrypt both to get $f(n)$ and $f(c)$ and apply f^{-1} to get n and c. Lastly, she can construct message 1 in $\mathbf{P1}$ or $\mathbf{P2}$ with c, n and $pk(b)$ to verify the guess.

Appendix B : Attack on Heather et al.'s scheme

Consider the Woo and Lam authentication protocol, $\pi 1$:

Msg 1. $a \rightarrow b : a$
Msg 2. $b \rightarrow a : nb$
Msg 3. $a \rightarrow b : \{a, b, nb\}_{sh(as)}$
Msg 4. $b \rightarrow s : \{a, b, \{a, b, nb\}_{sh(as)}\}_{sh(bs)}$
Msg 5. $s \rightarrow b : \{a, b, nb\}_{sh(bs)}$

$sh(xy)$ represents a shared-key between agents x and y. Heather et al. present a type-flaw attack on this protocol:

Msg 3. $a \rightarrow b : nb$
Msg 4. $b \rightarrow I_s : \{a, b, nb\}_{sh(bs)}$
Msg 5. $I_s \rightarrow b : \{a, b, nb\}_{sh(bs)}$

The attack works by (i) using a type-flaw in message 3 (nb in place of $\{a, b, nb\}_{sh(as)})\}$ and (ii) replay of message 4 in message 5. Heather et al. argue that inserting unique component numbers inside encryptions prevents this attack. In their scheme, the same protocol would be implemented as:

Msg 1. $a \rightarrow b : a$
Msg 2. $b \rightarrow a : nb$
Msg 3. $a \rightarrow b : \{a, b, nb, 1\}_{sh(as)}$
Msg 4. $b \rightarrow s : \{a, b, \{a, b, nb, 1\}_{sh(as)}, 2\}_{sh(bs)}$
Msg 5. $s \rightarrow b : \{a, b, nb, 3\}_{sh(bs)}$

However, Heather et al's results are valid *only* when assuming the standard inference rules. To see why, consider the inference rule $\{m, n\}_k \vdash \{m\}_k$ which would hold when using Cipher Block Chaining for encryption.

Msg 1. $a \rightarrow b : a$
Msg 2. $b \rightarrow a : nb$
Msg 3. $I(a) \rightarrow b : (nb, 3)$ /* In place of $\{a, b, nb, 1\}_{sh(as)}$ */
Msg 4. $b \rightarrow I(s) : \{a, b, (nb, 3), 2\}_{sh(bs)}$
Msg 5. $I(s) \rightarrow b : \{a, b, nb, 3\}_{sh(bs)}$ /* using CBC inf rule on Msg 4. */

This attack works because, an attacker can infer $\{a, b, nb, 3\}_{sh(bs)}$ from Msg 4 ($\{a, b, (nb, 3), 2\}_{sh(bs)}$) using the CBC inference rule.

Note that according to Heather et al., if there is an attack on a protocol using component numbering, there is also an attack on the protocol when using their original tagging scheme (although it is doubtful whether the result applies for inference rules outside the standard set).

Secure Protocols for Secrecy*

Hanane Houmani   Mohamed Mejri

LSFM Research Group
Computer Science Department
Laval University
Sainte-Foy, Qc, Canada
{mohamed.mejri,hanane.houmani}@ift.ulaval.ca

Abstract

This paper aims to ensure the correctness of cryptographic protocols with respect to the secrecy property. The idea is to establish some sufficient conditions (hypotheses) under which the correctness of a given protocol is guaranteed. Intuitively, a protocol is said to be correct with respect to the secrecy property if every valid trace (a trace in which all honest agents act according to the protocols specification and any message used by the intruder is previously defined) does not leak any sensitive information. To this end, we give some conditions that restrict what honest users participating in the protocol can send as messages.

Keywords: Cryptographic Protocols, Intruder, Role, Trace, Secrecy, Correctness.

1 Introduction

Today, it is well-known that the design of cryptographic protocols is error prone. Several protocols have been shown flawed in computer security literature [Car94, CJ96, KMM94, Lie93, Mea96, Syv92] many years after their publication. In spite of the interesting activities of research which led to correct a significant number of errors in the design of cryptographic protocols using different methods [Aba97, Bor01, DDMM02, Low98, Pau98, Sto01, THG99], the problem still not overcome and far from being controlled. This is due, on one hand, to the complexity and the subtlety of the cryptographic protocols themselves and, on the other hand, to the limitations of the current methods and techniques.

We believe, however, that the complexity related to the analysis of security protocols can be considerably reduced if we focus on some special classes of these protocols and limited security properties. This is the way that we have chosen to follow in this paper. In fact, the primary objective of this work is to define a set of correct cryptographic protocols with respect to the secrecy property. More precisely, in this paper we propose sufficient conditions that guarantee that a given cryptographic protocol does not leak any sensitive information. These conditions are three and aim to restrict the way according to which the principals involved in the analyzed protocol interact with each others. Intuitively, the first condition allows to ensure that all secret messages are encrypted by secret keys when they are exchanged. The second condition states that a principal that receives an initially unknown message cannot sent it as a component of other messages. However, he can use this received and initially unknown message as a key. The third condition forbid an honest principal to encrypt message m using a key k, if the key k appears inside the message m. Since we have proved that each protocol that satisfies these conditions is correct with respect to the secrecy property, then the checking of the correction of a protocol is an easy task. Indeed, to verify whether a protocol satisfies or not these conditions we proceed as follows:

- From the protocol, we extract what we call role-based specifications which aims to show how honest principal perceive the protocol.
- Once, the generalized roles are generated, we check whether they satisfy or not the three defined conditions. If true, we conclude that the protocol is correct with respect to the secrecy property.

*This research is supported by a research grant from the Natural Sciences and Engineering Council of Canada, NSERC, and the "Fonds Québécois de la Recherche sur la Nature et les Technologies", FQRNT.

The secrecy property is formalized using the notion of valid traces. Intuitively, a trace is an interleaving of many runs of the protocol in the presence of an active intruder. A trace is considered as valid when all the honest principals act according to the protocol specification and all the messages sent by the intruder are previously known by him. We consider a protocol to be correct, with respect to the secrecy property, if each one of its valid traces does not leak any secret information. The link between the three conditions and the secrecy property is implemented in a theorem which states that the correctness of the protocol with respect to this property is guaranteed whenever the conditions are respected.

The remainder of this paper is organized as follows: In Section 2, we explain how to model the protocol ,i.e., how to extract its generalized roles. In Section 3, we give a formalization of the secrecy property based on valid traces. Section 4 motivates and formalizes our three assumptions. Section 5 is reserved for the main result which is a theorem that ensure the correctness, with respect to the secrecy property, of all the protocols that respect the three assumptions. In Section 6, we give an example of correct protocol. Finally, in Section 7, some concluding remarks on this work and future research are ultimately sketched as a conclusion.

2 Protocol Modelling

In this section, we explain how to generate a role-based specification from protocol description written in the standard notation. These role-based specifications were first introduced in [DMTY97] for the purpose of verification of authentication protocols.

2.1 Protocol

Hereafter, we introduce the basic notations that will be used throughout this paper. A message is composed of one or more primitive (atomic) words. A message m encrypted with key k is written $\{m\}_k$ and forms a word by itself. Concatenated messages are separated by commas. Message contents (words) have the following naming conventions: Encryption keys and nonces are respectively written k and N. Principals are written A, B, S and I, where A and B stand for principals who wish to communicate, S for a trusted server and I for a potential intruder. Subscripts will be used to denote an association to a principal; thus, for example N_a is a nonce that belongs to A and k_{as} is a shared key between A and S. More formally, the set of messages can be defined by the following BNF grammar.

$$
\begin{array}{lcll}
m & ::= & A & \text{Principal Identifier} \\
 & | & T_a & \text{Timestamp} \\
 & | & N_a & \text{Nonce} \\
 & | & k & \text{Key} \\
 & | & X & \text{Message Variable} \\
 & | & \{m\}_k & \text{Encrypted Message} \\
 & | & m, m' & \text{Message Concatenation}
\end{array}
$$

Notice that we use $m.m'$ to mean concatenation if the notation m, m' introduces ambiguities especially when we manipulate sets of messages. Notice also that throughout this paper we use $\mathcal{M}$ to denote the set of all possible messages and $\mathcal{X}$ to denote the set of message variables.

As the message grammar shows, messages are constructed using atomic components (principal identifiers, nonces and keys) and two binary functions named concatenation and encryption. Given a set of messages M, the atomic components involved in the messages of M, denoted $M_{\Downarrow}$, can be constructively defined as follows:

Definition 2.1 (Flatness) *The flatness of a set of messages M, denoted by $M_{\Downarrow}$, is defined as the normal form obtained with the following rewriting system:*

- *Decrypt* : $(M \cup \{\{m\}_k\})_{\Downarrow} \rightarrow (M \cup \{m,\ k\})_{\Downarrow}$
- *Decompose* : $(M \cup \{m_1.m_2\})_{\Downarrow} \rightarrow (M \cup \{m_1, m_2\})_{\Downarrow}$

Now, we are ready to give a formal definition of a protocol. Basically, a protocol is specified by a sequence of communication steps given in the standard notation together with the initial knowledge (non-fresh and fresh knowledge) of each honest agent participating in this protocol as following:

Definition 2.2 *A protocol is defined by a pair $\langle P,\ K\rangle$, where:*

- *P is a sequence of communication steps. Each step has a unique identifier and specifies the sender, the receiver and the transmitted message. More precisely P has to respect the following BNF grammar:*

$$P \quad ::= \langle i,\ A \rightarrow B : m\rangle \mid P.P$$

The statement $\langle i,\ A \rightarrow B : m\rangle$ denotes the transmission of a message m from the principal A to the principal B in the step i of the protocol.

- *K is a set of triples (X, K_X, F_X) where X is an agent participating in the protocol, K_X is a set of non-fresh messages initially known by the agent X and F_X is a set of fresh messages (nonce, etc.) generated by X during the protocol.*

As an example, we give hereafter a protocol that achieves a key distribution between two principals A and B via a trusted server S.

Example 2.3 *Let p be the following protocol:*

$$
\begin{array}{lll}
p & = & \langle P, K\rangle \\
\textit{Where} & \left\{ \begin{array}{lll}
P & = & \langle 1, A \rightarrow S : \{A.B.N_a\}_{k_{as}}\rangle. \\
 & & \langle 2, S \rightarrow A : \{\{A\}_{N_a}.B.k_{ab}\}_{k_{as}}\rangle. \\
 & & \langle 3, S \rightarrow B : \{A.B.k_{ab}\}_{k_{bs}}\rangle \\
 & & \\
K & = & \{(A, K_A, F_A), (B, K_B, F_B), (S, K_S, F_S)\} \\
 & \textit{Where} & K_A = \{A, B, S, k_{as}\} \\
 & & K_B = \{A, B, S, k_{bs}\} \\
 & & K_S = \{A, B, S, k_{ab}, k_{bs}, k_{as}\} \\
 & & F_A = \{N_a\} \\
 & & F_B = \emptyset \\
 & & F_S = \{k_{ab}\}
\end{array} \right.
\end{array}
$$

Notice that for the sake of simplicity, we focus in this paper only on cryptographic protocols using symmetric keys. However, the same ideas can be applied for protocols which are based on asymmetric key encryption.

2.2 Roles and Generalized Roles

From a protocol, we extract what we call a role-based specification on which the assumptions have to be verified to check whether the protocol is correct with respect to the secrecy property or not. This role-based specification is a set of generalized roles extracted from the analyzed protocol. In the sequel, we define roles and generalized roles and we explain how we can extract them from a protocol.

A role is a protocol abstraction where the emphasis is put on a particular principal. More precisely, roles are extracted from the protocol according to the following steps:

- For each principal (agent), we extract from the protocol all the steps in which this principal participates. Furthermore, we add the same session identifier α to all those steps. Finally, to each fresh message, we add α as exponent to syntactically indicate that those messages change their values from one run of the protocol to another.

- We introduce explicitly an intruder I to point out that all sent and received messages by an honest agent are sent to (or received from) an intruder.

For instance, in the case of the protocol given by example 2.3, three roles, denoted $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{S}$, can be extracted. They respectively correspond to principals A, B and S and are defined as following:

$$\begin{array}{rcl} \mathcal{A} & = & \langle \alpha.1, A \rightarrow I(S) : \{A.B.N_a^\alpha\}_{k_{as}} \rangle. \\ & & \langle \alpha.2, I(S) \rightarrow A : \{\{A\}_{N_a^\alpha}.B.k_{ab}^\alpha\}_{k_{as}} \rangle \\ \\ \mathcal{B} & = & \langle \alpha.3, I(S) \rightarrow B : \{A.B.k_{ab}^\alpha\}_{k_{bs}} \rangle \\ \\ \mathcal{S} & = & \langle \alpha.1, I(A) \rightarrow S : \{A.B.N_a^\alpha\}_{k_{as}} \rangle. \\ & & \langle \alpha.2, S \rightarrow I(A) : \{\{A\}_{N_{a^\alpha}}.B.k_{ab}^\alpha\}_{k_{as}} \rangle. \\ & & \langle \alpha.3, S \rightarrow I(B) : \{A.B.k_{ab}^\alpha\}_{k_{bs}} \rangle \end{array}$$

From the roles, we extract what we call generalized roles. A generalized role is an abstraction of a role where some messages are replaced by variables. Intuitively, we replace a message or a component of message by a variable, if the receiver of this message could not do any verification on it. Generalized roles give a precise idea about the principals' behaviors during the protocol execution. For instance, the principal playing the role A receives at the second steps a previously unknown message (k_{ab}^α). If we suppose also that the exchanged messages are not typed (for example, there is no difference between nonce or any other message), then we conclude that A can accept any message instead of k_{ab}^α. This fact is captured by replacing the nonce k_{ab}^α by a variable X yielding to the following A's generalized role:

$$\mathcal{A}_G = \left\{ \begin{array}{l} \langle \alpha.1, A \rightarrow I(S) : \{A.B.N_a^\alpha\}_{k_{as}} \rangle. \\ \langle \alpha.2, I(S) \rightarrow A : \{\{A\}_{N_a^\alpha}.B.X\}_{k_{as}} \rangle \end{array} \right.$$

In the same way, we obtain B's generalized role and S's generalized role:

$$\mathcal{B}_G = \left\{ \ \langle \alpha.3, I(S) \rightarrow B : \{A.B.Y\}_{k_{bs}} \rangle \right.$$

$$\mathcal{S}_G = \left\{ \begin{array}{l} \langle \alpha.1, I(A) \rightarrow S : \{A.B.Z\}_{k_{as}} \rangle. \\ \langle \alpha.2, S \rightarrow I(A) : \{\{A\}_Z.B.k_{ab}^\alpha\}_{k_{as}} \rangle. \\ \langle \alpha.3, S \rightarrow I(B) : \{A.B.k_{ab}^\alpha\}_{k_{bs}} \rangle. \end{array} \right.$$

Notice that variables used in different roles are independent and then can be renamed.

In the rest of this paper, we use $\mathcal{R}(p)$ to denote all the roles of the protocol p and $\mathcal{R}_G(p)$ all it generalized roles. Besides, we use $\overline{\mathcal{R}(p)}$ (resp. $\overline{\mathcal{R}_G(p)}$) to denote the set of all the communication steps in all the roles in $\mathcal{R}(p)$ (resp. $\mathcal{R}_G(p)$).

3 Formalization of the Secrecy Property

Since the secrecy property has been used with different meanings in the literature [Aba97, AG96, Low96], then it is worth to clarify the signification associated to this term in this paper. Intuitively, a protocol keeps a message m secret, if there is no valid trace that leaks this message to an intruder. To formally introduce the secrecy property, we need first to define valid traces associated to a given protocol.

3.1 Valid Trace

Intuitively, a trace is said to be valid if all honest agents act according to the protocol specification and all the messages sent by the intruder are derivable from his knowledge. A message m is said to be derivable by the intruder from his knowledge, say K_I (a set of messages), whenever m can be obtained from K_I using the intruder's conventional abilities (encryption, decryption, composition, and decomposition). Furthermore, we consider that an honest agent acts according to the protocol specification if any given run in which he participates is an instance (variables are replaced by constant messages) of a prefix of his generalized role. The reason of taking into consideration even a prefix of a generalized role is that an honest agent may accidentally or voluntarily aborts the execution of a run of the protocol.
To formally introduce the notion of a valid trace of a given protocol, we need the following ingredients:

1. Trace: Intuitively, a trace is a sequence of communication steps when all the exchanged messages between honest agents pass via an intruder. More formally, a trace can be defined by the following BNF grammar:

$$\begin{array}{rcl} \tau & ::= & \varepsilon \\ & & \mid \langle \alpha.i,\ A \rightarrow\ I(B):\ m\rangle.\tau \\ & & \mid \langle \alpha.i,\ I(A) \rightarrow B:\ m\rangle.\tau \end{array}$$

Notice that if τ is a trace, then we denote by $\overline{\tau}$ the set of all its communication steps.

2. Session Identifiers: Each step in a given trace has a session identifier. The set of all session identifiers associated to the trace τ, denoted by $\mathcal{S}^{\tau}$, is defined as follows:

$$\begin{array}{rcl} \mathcal{S}^{\varepsilon} & = & \emptyset \\ \mathcal{S}^{\langle \alpha.i, A \rightarrow I(B):m\rangle.\tau} & = & \{\alpha\} \cup \mathcal{S}^{\tau} \\ \mathcal{S}^{\langle \alpha.i, I(A) \rightarrow B:m\rangle.\tau} & = & \{\alpha\} \cup \mathcal{S}^{\tau} \end{array}$$

3. Session: A trace is generally an interleaving of many sessions, where each of them has a unique identifier. We define τ^{α}, the session in the trace τ having α as identifier, as follows:

$$\begin{array}{rcll} (\varepsilon)^{\alpha} & = & \varepsilon & \\ (\langle \beta.i, A \rightarrow I(B):m\rangle.\tau)^{\alpha} & = & \tau^{\alpha} & \text{if } \alpha \neq \beta \\ (\langle \beta.i, I(A) \rightarrow B:m\rangle.\tau)^{\alpha} & = & \tau^{\alpha} & \text{if } \alpha \neq \beta \\ (\langle \alpha.i, A \rightarrow I(B):m\rangle.\tau)^{\alpha} & = & \langle \alpha.1, A \rightarrow I(B):m\rangle.\tau^{\alpha} & \\ (\langle \alpha.i, I(A) \rightarrow B:m\rangle.\tau)^{\alpha} & = & \langle \alpha.1, I(A) \rightarrow B:m\rangle.\tau^{\alpha} & \end{array}$$

Notice that ε is an empty trace and can be eliminated if it is found within an non empty one.

4. Def/Use: In a given trace, the intruder knowledge changes from one step to another. Suppose that K_I is the initial intruder's knowledge, then the defined (known) messages by the intruder after the execution of a trace τ, denoted by $\mathsf{Def}_{K_I}(\tau)$, is introduced as follows:

$$\begin{array}{rcl} \mathsf{Def}_{K_I}(\varepsilon) & = & K_I \\ \mathsf{Def}_{K_I}(\langle \alpha.i, A \rightarrow I(B):m\rangle.\tau) & = & \{m\} \cup \mathsf{Def}_{K_I}(\tau) \\ \mathsf{Def}_{K_I}(\langle \alpha.i, I(B) \rightarrow A:m\rangle.\tau) & = & \mathsf{Def}_{K_I}(\tau) \end{array}$$

Moreover, in a given trace τ, the intruder uses some messages which are in the set $\mathsf{Use}_{K_I}(\tau)$ that is defined as follows:

$$\begin{array}{rcl} \mathsf{Use}_{K_I}(\varepsilon) & = & K_I \\ \mathsf{Use}_{K_I}(\langle \alpha.i, A \rightarrow I(B):m\rangle.\tau) & = & \mathsf{Use}_{K_I}(\tau) \\ \mathsf{Use}_{K_I}(\langle \alpha.i, I(B) \rightarrow A:m\rangle.\tau) & = & \{m\} \cup \mathsf{Use}_{K_I}(\tau) \end{array}$$

The Def/Use notions are very important elements to define valid traces. In fact, in a valid trace all messages used by the intruder have to be previously defined by him. For the sake of simplicity, we use in the sequel $\mathsf{Def}(\tau)$ (resp. $\mathsf{Use}(\tau)$) to denote $\mathsf{Def}_{K_I}(\tau)$ (resp. $\mathsf{Use}_{K_I}(\tau)$) when K_I is clear from the context.

We also introduce the reduction operator to formalize the notion of valid trace as follows:

Definition 3.1
The reduction of a set of messages M, denoted by $M_{\downarrow}$, is defined as the normal form[1] *of M obtained from the following rewriting rules:*

- *Decryption* : $(M \cup \{\{m\}_k,\ k\})_{\downarrow} \rightarrow_e (M \cup \{m,\ k\})_{\downarrow}$
- *Decomposition :* $(M \cup \{m_1.m_2\})_{\downarrow} \rightarrow_c (M \cup \{m_1, m_2\})_{\downarrow}$

[1] We mean here that the application of any rewriting rule does not bring modification to M

5. Well-Defined Trace: Intuitively, a trace is considered to be well-defined if all messages used by the intruder are previously defined by him. More formally, a trace τ is said to be well defined if for all traces τ_1 and τ_2 and communication step e such that $\tau = \tau_1.e.\tau_2$, we have:

$$\mathsf{Use(e)}_{\downarrow} \subseteq \mathsf{Def}(\tau_1)_{\downarrow}$$

6. Well-Formed Trace: Intuitively, a trace is well formed for the protocol p if all the honest agents participating in it act according to the protocol. More formally, let p be a protocol. We say that a trace τ is well formed for the protocol p, or p-well formed, if for all session identifier α in $\mathcal{S}^{\tau}$, there exist r in $\mathcal{R}_G(p)$, r_1 a prefix of r and a substitution σ such that:
$$\tau^{\alpha} = r_1\sigma$$

7. Valid Trace: Now, we can easily define a valid trace. In fact a trace τ is valid for a given protocol p, or p-valid, if the two following conditions hold:
   - τ is well-defined.
   - τ is p-well formed.

In the sequel, we use $\mathcal{D}(p)$ to denote all the messages sent within all the generalized roles of p. More formally :

$$\mathcal{D}(p) = \bigcup_{e_i \in \overline{\mathcal{R}_G(p)}} \mathsf{Def}(e_i)$$

3.2 Secrecy Property

Hereafter, we give a formal definition of the secrecy property.

Definition 3.2 (Secrecy Property) *Let p be a protocol and S a set of atomic messages. We say that p does not leak any information in S (i.e., p is correct with respect to the secrecy property) if for any valid trace τ of p, we have:*

$$S \cap \mathsf{Def}(\tau)_{\downarrow} = \emptyset$$

In the sequel, we denote by $\mathcal{SP}(S)$ the set of all protocols which are correct with respect to the secrecy property for a given set S of secret messages.

4 Assumptions

This section introduces and motivates the basic assumptions used to guarantee the correctness of a cryptographic protocol with respect to the secrecy property. Obviously, it is worthwhile that the proposed hypothesis can be easily verified within any protocol given in its standard notation.

Zero-Unprotected Secret Message This assumption states that any secret message exchanged during the protocol has to be encrypted using a secret key. Obviously, this assumption is necessary but it is far from being sufficient. In fact, even if it is respected, secrecy flaws may persist.

To ensure this condition, it is sufficient to gather all the messages sent by honest agents and to verify if the intruder is able to deduce from them any secret message by using his usual abilities. To formally introduce this assumption, we need the following definition:

Definition 4.1 (Extended Reduction) *Let M be a set of messages. The extended reduction of M, denoted by $M_{\downarrow_x}$, is defined as the normal form of M obtained using the following rewriting rules:*

$$\begin{array}{lll} M \cup \{m_1.m_2\} & \rightarrow_{c_x} & M \cup \{m_1, m_2\} \\ M \cup \{\{m\}_{\alpha}, \beta\} & \rightarrow_{e_x} & M \cup \{\{m\}_{\alpha}, \beta\} \cup \{m\sigma, \beta\sigma \mid \sigma = mgu(\alpha, \beta)\} \end{array}$$

We have proved that under some conditions this rewriting rules converge (for detailed proof, contact the authors). Now, the first assumption can be stated as follows:

Assumption 4.1 *We say that a protocol p satisfies the assumption of zero-unprotected secret message with respect to a set of secret messages S, if the following condition holds.*

$$S \cap \mathcal{D}(p)_{\downarrow_x} = \emptyset$$

In the sequel, we denote by $\mathcal{H}_1(S)$ the set of all protocols that satisfy the assumption of zero-unprotected secret message with respect to a set of secret messages S.

Zero-Unknown Sent Message Intuitively, this hypothesis forbids an honest agent to send an unknown message (a message that he has received from another agent but he cannot do any verification on its value) either in clear or encrypted. However, unknown messages can be used as keys to encrypt messages and eventually send them.

This second assumption allows us to control what an honest agent can say during any run of the analyzed protocol. This control is very important since whenever an honest agent talks (sends a message) he risks to generate problem. In fact, honest agents detain secret information and whenever they speak, they risk to say some thing that allows an intruder to deduce one of these secrets.

To formally introduce this assumption, we need the following definition that allows us to extract all atomic components, except keys, that can be found in a set of messages.

Definition 4.2 ($\mathcal{V}^-$) *The function $\mathcal{V}^-$ is defined as follows:*

$$\mathcal{V}^- : 2^{\mathcal{M}} \quad \rightarrow \quad 2^{\mathcal{M}}$$

Such that :

$$\begin{array}{rcl}
\mathcal{V}^-(M_1 \cup M_2) & = & \mathcal{V}^-(M_1) \cup \mathcal{V}^-(M_2) \\
\mathcal{V}^-(\{\{m\}_k\}) & = & \mathcal{V}^-(\{m\}) \\
\mathcal{V}^-(\{m_1.m_2\}) & = & \mathcal{V}^-(\{m_1\}) \, \cup \, \mathcal{V}^-(\{m_2\}) \\
\mathcal{V}^-(\{m\}) & = & \{m\} \textit{ if } m \textit{ is atomic}
\end{array}$$

Now, the second assumption can be formalized as follows:

Assumption 4.2 *We say that a protocol p satisfies the assumption of zero-unknown sent message, if the following condition holds.*

$$\mathcal{X} \cap \mathcal{V}^-(\mathcal{D}(p)) = \emptyset$$

It is important to notice that this second assumption does not exclude the use of nonces. In fact, a nonce can be used as a key to encrypt a known message. For instance, as shown later the protocole of example 2.3 respects this assumption.
In the sequel, we denote by $\mathcal{H}_2$ the set of all protocols that satisfy this second assumption.

Key Restriction This final assumption states that a key used to encrypt a message m cannot be a component of m. The following definition makes easy the formalization of this final assumption. In fact, it defines a function that allows to know if a key used to encrypt a given message can be found inside this message.

Definition 4.3 (F) *Let F be the following function:*

$$F : 2^{\mathcal{M}} \quad \rightarrow \quad \{True, False\}$$

Such that :

$$\begin{array}{rcl}
F(M_1 \cup M_2) & = & F(M_1) \wedge F(M_2) \\
F(\{\{m\}_k\}) & = & (k \not\in \{m\}_{\Downarrow}) \, \wedge \, F(\{m\}) \\
F(\{m_1.m_2\}) & = & F(\{m_1\}) \, \wedge \, F(\{m_2\}) \\
F(\{m\}) & = & True \textit{ if } m \textit{ is atomic}
\end{array}$$

This assumption is basically introduced to simplify proofs and we believe that it can be removed. The formalization of this assumption is as follows :

Assumption 4.3 *We say that a protocol p satisfies the assumption of key restriction, or $p \in \mathcal{H}_3$, if the following condition holds.*

$$F(\mathcal{D}(p)) = true$$

In the sequel, we denote by $\mathcal{H}_3$ the set of all protocols that satisfy this third assumption.

5 Main Result

The main result of this paper is given by the following theorem which states that any protocol that respects the three defined assumptions given above is correct with respect to the secrecy property.

Theorem 5.1 $\mathcal{H}_1(S) \cap \mathcal{H}_2 \cap \mathcal{H}_3 \subseteq \mathcal{SP}(S)$

Proof:

The main idea behind the proof can be sketched (detailed proof was removed due to the space limitation) as following:

Suppose that for a given protocol p, we can define a set of messages M_p such that if a valid trace of p allows an intruder to discover a secret message m then there exists a substitution σ such that $m \in M_p\sigma$. Now, since we suppose that secret messages are atomic, then if m is secret (atomic) and $m \in M_p\sigma$, it follows that $s \in M_p$ or there exists a variable x such that $x \in M_p$. Therefore, to ensure the correctness of the protocol p, with respect to the secrecy property, it will be sufficient to impose conditions that forbid variables and secret messages to be in M_p. We prove that $M_p = \mathcal{D}(p)_{\downarrow_x}$ where $\mathcal{D}(p)$ is the set of messages sent by honest agents within their generalized roles. The assumptions, on the other hand, contribute as follows:

- The assumption $\mathcal{H}_1(\{s\})$ ensures that $s \notin \mathcal{D}(p)_{\downarrow_x}$.
- The restriction $\mathcal{H}_2$ guarantees that the set $\mathcal{D}(p)_{\downarrow_x}$ does not contain any variable.
- Finally, the hypothesis $\mathcal{H}_3$ helps to easily prove the existence of the set $\mathcal{D}(p)_{\downarrow_x}$.

□

6 Case Study

In the sequel, we show that the protocol of example 2.3 respects the three assumption and then we conclude that it is with respect to the secrecy property.
Let $K_I = \{A, B, S, k_{is}, k_{ib}^\alpha, k_{ai}^\alpha, N_i^\alpha\}$ be the initial knowledge of the intruder, then from the generalized roles given in section 2 we deduce that:

$$\mathcal{D}(p) = K_I \cup \{\{A.B.N_a^\alpha\}_{k_{as}}, \{\{A\}_Z.B.k_{ab}^\alpha\}_{k_{as}}, \{A.B.k_{ab}^\alpha\}_{k_{bs}}\}$$

From definition 4.1, we deduce that:

$$\mathcal{D}(p)_{\downarrow_x} = K_I \cup \{\{A.B.N_a^\alpha\}_{k_{as}}, \{\{A\}_Z.B.k_{ab}^\alpha\}_{k_{as}}, \{A.B.k_{ab}^\alpha\}_{k_{bs}}\}$$

Let, for instance, $S = \{k_{ab}^\alpha\}$ be the set of secret messages.

- **Verification of the first assumption:** We have:

  $$\mathcal{D}(p)_{\downarrow_x} \cap S = \emptyset$$

  We deduce that this protocol satisfies the assumption of zero-unprotected secret message.

- **Verification of the second assumption:** Since:

  $$\mathcal{V}^-(\mathcal{D}(p)) = K_I \cup \{k_{ab}^\alpha\}$$

  We deduce that the protocol satisfies the assumption of zero-unknown sent message.

- **Verification of the third assumption:** We have $F(\mathcal{D}(p)) = True$, then the protocol satisfies the assumption of key-restriction.

Since the protocol satisfies the tree assumptions, then we conclude that the protocol is correct with respect to the secrecy property.

7 Conclusion

It is well known that the design of cryptographic protocols is error prone due, on one hand, to the complexity and the subtlety of the cryptographic protocols themselves and, on the other hand, to the limitations of the current methods and techniques. The complexity of this problem can be considerably reduced if we restrict the set of protocols that we want to analyze. This paper is a contribution in this respect with the following items:

- It introduces necessary conditions that ensure the correctness of security protocols with respect to the secrecy property. These conditions can be directly verified on generalized roles that can be easily extracted from the analyzed protocol. The aim of these assumptions is to restrict the way according to which honest agents interact with each other in order to reduce the intruder abilities (especially those abilities given by the protocol itself).
- It gives an exemple of correct protocol with respect to the secrecy property, i.e., a protocol that respects the defined assumptions. We believe also that there are protocols which are correct with respect to the secrecy property and do not respect one of these hypotheses. In other words, we believe that we can ensure the correctness with more relaxed assumptions.
- It point out, due to the second assumption ($\mathcal{H}_2$), an important principle for the design of cryptographic protocols whatever are their goals. In fact, this assumption aims to restrict considerably the number of variables generated within generalized roles which involves the reduction of the intruder abilities and consequently it reduces the risk of flaws.

As a future works, we plan to deeply study the assumptions and more interestingly to investigate other security properties (integrity, authentication, etc.) and other class of protocols.

Bibliography

[Aba97] M. Abadi. Secrecy by Typing in Security Protocols. In *Theoretical Aspects of Computer Software: Third International Conference, TACS'97 Proceedings*. Lecture Notes in Computer Science, page 611-638. Springer Verlag, September 1997.

[AG96] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. Technical report, DEC/SRC, December 1996.

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. *Lecture Notes in Computer Science*, 2076:667–681, 2001.

[Car94] U. Carlsen. *Formal Specification and Analysis of Cryptographic Protocols*. PhD thesis, Thèse d'Informatique soutenue à l'Université PARIS XI, October 1994.

[CJ96] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature. Unpublished Article Available at `http://dcpu1.cs.york.ac.uk/~jeremy`, August 1996.

[DDMM02] M. Debbabi, N. Durgin, M. Mejri, and J. Mitchell. Security by Typing. *the International Journal on Software Tools For Technology Transfer (STTT); DOI 10.1007/s10009-002-0100-7, Springer Verlag*, 2002.

[DMTY97] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A New Algorithm for Automatic Verification of Authentication Cryptographic Protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols, DIMACS Center, Core Building, Rutgers University, New Jersy, USA*, Sep 1997.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three Systems for Cryptographic Protocol Analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[Lie93] A. Liebl. Authentication in Distributed Systems: A Bibliography. *Operating Systems Review*, 27(4):122–136, October 1993.

[Low96] G. Lowe. Breaking and fixing the needham schroeder public-key protocol using fdr. In *Proceedings of TACAS*, volume 1055, pages 147–166. Springer Verlag, 1996.

[Low98] G. Lowe. Towards a Completeness Result for Model Checking of Security Protocols. . *Proceedings of 11th IEEE Computer Security Foundations Workshop,*, pages 96–108, 1998.

[Mea96] C. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Proceedings of Asiacrypt 96*, 1996.

[Pau98] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[Sto01] S. D. Stoller. A bound on attacks on payment protocols. In *Logic in Computer Science*, pages 61–70, 2001.

[Syv92] P. Syverson. Knowledge, Belief, and Semantics in the Analysis of Cryptographic Protocols. *Journal of Computer Security*, 1(3):317–334, 92.

[THG99] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct, 1999.

Part III

Invited Talk

Privacy in Today's World: Solutions and Challenges

Rebecca Wright
Stevens Institute of Technology
Hoboken, NJ — USA
rwright@cs.stevens-tech.edu

Abstract

Privacy is being increasingly eroded as technological advances make it possible to capture, store, and process large amounts of personal data. In this talk, I will overview the current privacy landscape and outline some of the important problems in this area. I will also describe some of my work in this area, on privacy-protecting statistical analysis.

Suppose a client wishes to compute some aggregate statistics on a privately-owned data base. The data owner wants to protect the privacy of the personal information in the data base, while the client does not want to reveal his selection criteria. Privacy-protecting statistical analysis allows the client and data owner to interact in such a way that the client learns the desired aggregate statistics, but does not learn anything further about the data; the data owner leans nothing about the client's query.

Motivated by this application, we consider the more general problem of "selective private function evaluation," in which a client can privately compute an arbitrary function over a database. We present various approaches for constructing efficient selective private function evaluation protocols, both for the general problem and for privacy-protecting statistical analysis.

Part IV

Low-Level Primitives

Oblivious Comparator and its Application to Secure Auction Protocol

Hiroaki Kikuchi

Department of Information media technology
Tokai University
1117 Kitakaname, Hiratsuka, Kanagawa, 259-1292 Japan
kikn@tokai.ac.jp

Abstract

This paper presents a protocol for Secure Function Evaluation (SFE) in which n players have secret inputs $E[a_1]$, $E[a_2], \ldots, E[a_n]$, of a known boolean function f, and they collaborate to compute the ciphertext of the output of the boolean function, $E[f(a_1, \ldots, a_n)]$. The main result is a completeness theorem (Theorem 3.1) which states that an arbitrary function can be evaluated at the oblivious party without help of private information. The proposed protocol is based on the Jakobsson and Juels's Mix-and-Match scheme [JJ00] in which the truth table of a target function is row-wise randomized (*mixing*) using a Mix network, and then players perform "*matching*" the designated output ciphertext and the corresponding rows. The biggest difference between the proposed SFE and the Mix-and-Match is that the proposed protocol does not require any involvement of key holders to evaluate function, while the Mix-and-Match needs key holders to perform threshold decryptions at every step of evaluation of boolean gates. One disadvantage of the proposed scheme is the Reed-Muller expansion [Sas97] involves an exponential blow-up in the number of input, n, as the same as the conventional schemes, e.g., *CyptoComputer* proposed by Sander, Young, and Yung [SYY99]. This paper presents an efficient construction for a primitive called '*oblivious comparator*' with n-round complexity between the comparator and n players but the bandwidth spent by one communication is independent from n (linear to the size of values to be compared), and hence it does not suffer the blow-up in n. The oblivious comparator is suitable to implement a secure auction because an auctioneer communicates with bidders once at time, and performs evaluation without help of trusted key holders. In addition, the proposed construction allows arbitrary complicated functions including a search for second highest, a resolution the winner, and a dynamic programming (for combinatorial auction).

1 Introduction

1.1 Secure Auction

Auctions in the electronic commerce are very complicated. The simplest auction style is the open-bid English auction, in which bidders incrementally raise the prices bid for goods until as many winners are left as the number of units of goods. As an alternative to this classical style of auction, an automatic agent system called "proxy bidding" [eBay] is becoming popular. A Dutch-style auction naturally satisfies the property that privacy of losing bids is preserved after auction closes. The Vickery auction, in which the winner who has the highest bid pays the second highest bid.

In the theory of economics, it is known that a social surplus is maximized when a bidder whose evaluation is highest wins the auction game and pay the uniform winning price which is independent of their evaluation. Wurman et al. proved that the $(M+1)$st-price auction satisfies a useful property, *incentive compatibility*, i.e., the dominant strategy is for a bidder to bid to his/her true valuation [WWW98]. Since a winner's payment will be determined by the $(M+1)$st highest bid, which is the highest of all losing bids, every bidder who agrees to bid the maximum price he/she is willing to pay for a given item maximizes his/her chance to win without being worried that he/she might bid too much.

In order to satisfy the complicated requirements in secure auction, several attempts have been done in using cryptographical techniques. Franklin and Reiter present a sealed-bid auction protocol in [FR96]. The protocol uses a verifiable signature sharing in order to prevent malicious bidder from canceling their bids. Bids are kept secret until the opening phase, and then all bids are opened and compared to determine the highest one.

Kikuchi, Hakavy and Tygar [KHT99] improve the privacy of bids among distributed auctioneers even after the opening phase comes using a secure function computation of summation. The protocol runs in linear time to the number of possible bidding prices and cannot deal with tie breaking. In [Sak00], Sako implements a Dutch-style auction using a distributed decryption. In the protocol, a bidder casts his bid encrypted by the public key corresponding to his bidding price. The privacy of losers' bits are kept under the assumption of not all auctioneers being faulty. Similarly, Miyazaki and Sakurai use an undeniable signature [MS99], and Kobayashi and Morita use an one-way hash chain [KM99].

Auctions in the electronic commerce are more complicated. Multiple buyers and sellers are involved and multiple unit of goods are auctioned in several environments. Wurman, Walsh and Wellman examined a several auction designs and analyzed in terms of the incentive compatibility in [WWW98]. They showed that the $(M + 1)$st-price sealed-bid auction is incentive compatible for single-unit buyers. The secure second-price ($M = 1$) auction protocol is presented by Hakavy et al. in [HTK98]. They use the secure multiparty protocol of multiplication, presented in [BGW87] in order to resolve the second highest bid in $O(\log(k))$ rounds. Other papers on this subject include [NPS99, SA99, Cac99, SS99, Kud98, WI00].

1.2 Secure Function Evaluation

The best way to securely implement arbitrary auction style is a Secure Function Evaluation (SFE). A SFE is a protocol to allow Alice and Bob, having inputs a and b, respectively, to compute a known function $f(a, b)$ while keeping their inputs private.

Goldreich, Micali, and Wigderson [GMW87] presented a generalized scheme with an assumption of one-way trapdoor permutation, now commonly known as a secure multi-party computation (MPC). In their scheme, a target function is decomposed into gates represented as randomly permuted truth tables. Using a 1-2 oblivious transfer protocol, the other party picks the row of the truth table designated by his secret input bit. Although a target function is generally and systematically implemented using conventional circuit design technologies, intensive communication takes place to evaluate the function. The MPC scheme presented by [CDN01], which is one of the most efficient MPCs, requires $O(nk|C|)$ bits for broadcasting and $O(d)$ rounds for evaluation, where n is the number of parties, k is a security parameter and $|C|$ is the size of circuit.

Following the work by [GMW87], a number of MPC schemes have been proposed. Ben-Or, Goldwasser and Wigderson presented an information-theoretically MPC scheme, in which an arbitrary function is realized as a composition of two arithmetical operations, addition and multiplication [BGW87].

With the help of a cryptographic primitive known as verifiable secret sharing (VSS) [Ped91], robustness against active adversaries is assured. One advantage of their approach compared with that of [GMW87] is efficient field operations rather than bitwise manipulations. A drawback is more assumptions, such as VSS, necessary to achieve robustness. Recently, Cramer, Damgård and Maurer showed that any linear secret sharing scheme allows general construction of an MPC scheme [CDM00].

The Mix-and-Match scheme proposed by Jakobsson and Juels [JJ00] is a new approach to SFE. In their scheme, rather than sharing private inputs into some players, ciphertext manipulation is used to evaluate the function. A brief description is as follows. A player provides ciphertexts of his private input bit. A target function f is decomposed into Boolean gates that are represented by a truth table, in which entries are publicly provided ciphertexts.

There are two steps, mixing and matching. In *mixing*, a row-wise permutation of the truth table is computed via a conventional mix network (e.g., [Abe00]), and then players perform *matching*, in which some trusted parties look up the corresponding row in the mixed truth table and finally, without decrypting, the designated output ciphertext is obtained.

The interesting feature of the Mix-and-Match protocol is that only the key generation is distributed among trusted parties and an input does not have to be shared. The players do not even necessarily perform VSS and are not concerned about the other players. This feature makes performing the protocol simple and effective. Moreover, the target function can be easily constructed by boolean formulas as in the [GMW87] approach.

A disadvantage of the Mix-and-Match protocol is a cryptographic primitive called a plaintext equality test ($\mathcal{PET}$), which allows players to determine whether two given ciphertexts represent the same plaintext. To perform the $\mathcal{PET}$, the all member of trusted authorities, having piece of distributed decryption key, have to jointly participate in the threshold decryption process. Moreover, the $\mathcal{PET}$ is required for each evaluation of bit in a boolean function to be evaluated and thus it spends bandwidth so much.

1.3 Our Contribution

In this paper, we propose a new SFE scheme based on the Mix-and-Match ciphertext manipulation approach, but more suitable for practical secure auction. Rather than evaluating with the help of key holders, our protocol has an oblivious computer evaluate a target function without any knowledge of the private information. Therefore, every step of the computation can be made publicly verifiable; i.e., just showing all inputs and outputs, any party can ensure that the computer performs the correct manipulation. To prevent a malicious parity from misbehaving, some proof of knowledge protocols are used.

The main contribution of our paper is to prove that an arbitrary Boolean function can be evaluated without decrypting the input ciphertext within n rounds of iteration with players only decrypting once each. The well-known Boolean function canonical form called the Reed-Muller Expansion [Sas97] is introduced. There is a disadvantage of use of the Reed-Muller expansion, namely, it involves an exponential blow-up in the number of players n. In order to overcome the issue of blow-up, we present an efficient construction for a primitive called '*oblivious comparator*' with n-round complexity between the comparator and n players but the bandwidth spent by one communication is independent from n (linear to the size of values to be compared), and hence it does not suffer the blow-up in n. The oblivious comparator is suitable to implement a secure auction because an auctioneer communicates with bidders once at time, and performs evaluation without help of trusted key holders. In addition, the proposed construction allows arbitrary complicated functions including a search for second highest, a resolution the winner, and a dynamic programming (for combinatorial auction).

The outline of this paper is as follows. We generally describe a model and some building blocks in Section 2. In Section 3, we give the detail of our proposed protocol. As a practical application of our protocol, in Section 4, we present the *oblivious comparator*, which given n ciphertexts representing k-bit numbers generates the output ciphertexts in which the highest number and the identity of whose number has the highest number are encrypted. After computation at oblivious comparator, with the help of private key holders, the outputs are finally decrypted. The oblivious comparator cannot cheat bidders because of the proof of knowledge that she correctly performs comparison of bids but learn any knowledge from the result of the computation, namely, she does not know who is the winner nor how high the winning price is.

2 Model and Building Blocks

2.1 Model

We consider n players, $P_1, \ldots, P_n$, and some of these may be malicious. Each player a_i has a secret input a_i. A *computer* $\mathcal{C}$ takes n input ciphertexts, $E[a_1], \ldots, E[a_n]$, and outputs a ciphertext $E[y]$ of an agreed n-variable boolean function $y = f(a_1, \ldots, a_n)$. We assume a secure channel with confidentiality, sender authentication and message integrity between all pairs of the parties. We does not assume an authenticated broadcast channel in this model.

A computer $\mathcal{C}$ has a state S, which is a set of ciphertexts, and a state transition algorithm T, which takes an input ciphertext sent from a player and updates the state S according to the agreed function f. Every time $\mathcal{C}$ communicates with a player, the state S_i is updated to S_{i+1} and then referred by the subsequent player who uses the state to generate the next input ciphertext. Players communicate with $\mathcal{C}$ once, therefore, n rounds are involved to produce the final state S_n. After exactly n rounds between $\mathcal{C}$ and $P_1, \ldots, P_n$, $\mathcal{C}$ publishes the output ciphertext Y defined by a decoding algorithm D.

2.2 $\oplus$-Homomorphic Encryption Scheme

Let M be a set of plaintext, $\{m_0, m_1\}$, where m_0 and m_1 mean boolean values corresponding to 'false' and 'true', respectively.

Let us suppose a *homomorphic encryption scheme* E which satisfies the following properties:

- E is $\oplus$-homomorphic over $GF(2)$, i.e., for elements a and b of $\{m_0, m_1\}$,

$$E[a \oplus b] = E[a] \times E[b] \tag{1}$$

  holds, where $a \oplus b$ is an Exclusive OR, defined as $m_0 \oplus m_1 = m_1 \oplus m_0 = m_1$ and $m_0 \oplus m_0 = m_1 \oplus m_1 = m_0$.

- E is semantically secure, that is, no one is able to distinguish ciphertexts of m_0 and m_1 with probability significantly greater than a random guess.

– The key generation can be distributed among a certain number of players. The size of the public key should not depend on the number of shares.

– The decryption process can be distributed among t-out-of-n players who share the corresponding private key. The computational and communicational (bandwidth and rounds) costs should be as small as possible.

The El Gamal encryption satisfies the all requirements under the Decision Diffie-Hellman (DDH) assumption, if we let $m_0 = 1$ and $m_1 = -1 \pmod p$. Let p and q be large primes such that $p = 2q + 1$ and $\mathcal{G}$ be the set of multiplicative groups of order q in Z_p^*. Let g be a primitive element of $\mathcal{G}$.

An El Gamal encryption of message m with public key $y = g^x$ is of the form $E_a[m] = (M, G) = (my^a, g^a)$, where a is a random number chosen from Z_q. To decrypt the ciphertext (M, G), we use the corresponding private key x to compute $M/G^x = mg^{xa-ax} = m$. By element-wise multiplication, we define $E[a] \times E[b] = (M_a M_b, G_a G_b)$, which yields a new ciphertext $E[a \oplus b]$, and it can be seen that the El Gamal encryption is $\oplus$-homomorphic over $GF(2)$.

Distributed decryption is also feasible. A private key is jointly generated by the collaboration of t honest parties (key holders) out of n and distributed among them using $(t-1)$-degree random polynomials $f(x)$ as $f(1), f(2), \ldots, f(n)$. To decrypt a ciphertext provided with the public key $y = g^{f(0)}$, the i-th party publishes $G^{f(i)}$ for $i = 1, \ldots, t$, and then computes $G^{f(1)\gamma_1} \cdots G^{f(t)\gamma_t} = G^{f(0)}$ where γ_i is the LaGrange coefficient for i. For verifiability the players use Verifiable Secret Sharing (VSS) as proof of possession of $f(i)$ such that $G^{f(i)}$. See [Ped91] for details.

An other instance of homomorphic encryption scheme is QR encryption as used in [KMO01, GM84].

2.3 Proof of Knowledge

We will use a proof of knowledge of private input to the computer, which is based on the disjunctive and conjunctive proofs of knowledge in [CDS94].

Conjunctive Proof of Knowledge

By $PK\{(\alpha) : y_1 = g_1^\alpha \wedge y_2 = g_2^\alpha\}$, we denote a proof of knowledge of discrete logarithms of elements y_1 and y_2 to the bases g_1 and g_2. Selecting random numbers r_1 and $r_2 \in Z_q$, a prover sends $t_1 = g_1^{r_1}$ and $t_2 = g_2^{r_2}$ to a verifier, who then sends back a random challenge $c \in \{0,1\}^k$. The prover shows $s = r - cx \pmod q$, which should satisfy both $g_1^s y_1^c = t_1$ and $g_2^s y_2^c = t_2$.

Disjunctive Proof of Knowledge

We denote by $PK\{(\alpha, \beta) : y_1 = g^\alpha \vee y_2 = g^\beta\}$ to mean a proof of knowledge of one out of the two discrete logarithms of y_1 and y_2 to the base g. Namely, the prover can prove that he knows a secret value under which either $y = y_1$ or $y = y_2$ must hold without revealing which identity was used. Without loss of generality, we assume that the prover knows α for which $y = g^\alpha$ holds. The prover uniformly picks $r_1, s_2 \in Z_q$ and $c_2 \in \{0,1\}^k$ and sends $t_1 = g^{r_1}$ and $t_2 = g^{s_2} y_2^{c_2}$ to the verifier, who then gives a random challenge $c \in \{0,1\}^k$, where k is a security parameter. On receiving the challenge, the prover sends $s_1 = r_1 - c_1\alpha \pmod q$, s_2, c_1 and c_2, where $c = c_1 \oplus c_2$. The verifier can see if the prover is likely to have the knowledge by testing both $t_1 = g^{s_1} y_1^{c_1}$ and $t_2 = g^{s_2} y_2^{c_2}$ with provability $1 - 2^{-k}$. Note that the same test can be used when t_1 and t_2 are prepared for the other knowledge β.

2.4 Reed-Muller Expansion

Let us review an ordinary version of Reed-Muller exapnasion defined with boolean values before we go to the cryptographic expansion.

Lemma 2.1 *Let x, y and z be boolean values. Then, we have*

1. $x \oplus y = y \oplus x$,
2. $0 \oplus x = x, 1 \oplus x = \overline{x}$,
3. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$,
4. $a(x \oplus y) = ax \oplus ay$,
5. $x \vee y = x \oplus y \oplus xy$.

Lemma 2.2 (Shannon Expansion) *[Sas97] Let $f(x_1, \dots, x_n)$ be an n-variable boolean function. Then, f is expanded by*

$$\begin{aligned} f &= \overline{x_1}F_0 \vee xF_1 \\ &= \overline{x_1}F_0 \oplus xF_1 \\ &= F_0 \oplus x_1(F_0 \oplus F_1) \end{aligned} \tag{2}$$

where $F_0 = f(0, x_2, \dots, x_n)$ and $F_1 = f(1, x_2, \dots, x_n)$.

By recursively applying Eq.(2) for every variable in f, we have the following canonical form of f.

Lemma 2.3 (Reed-Muller Expression) *[Sas97] An arbitrary n-variable function $f(x_1, \dots, x_n)$ is represented as*

$$\begin{aligned} f = \; & a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n \\ & \oplus a_{12}x_1x_2 \oplus a_{13}x_1x_3 \oplus \cdots \oplus a_{n-1}a_nx_{n-1}x_n \\ & \vdots \\ & \oplus a_{12\cdots n}x_1x_2\cdots x_n. \end{aligned} \tag{3}$$

Given a function, the boolean coefficients $a_1, a_2, \dots, a_{12\cdots n}$ are uniquely determined. Equation (3) is called the Reed-Muller expression of f.

3 The Scheme

3.1 Logical Operations

We show that fundamental logical operations, conjunction ($\wedge$), disjunction ($\vee$) and negation ($\overline{x}$), are possible for input ciphertexts without decrypting.

Lemma 3.1 (negation) *Let $E[a]$ be a ciphertext of a homomorphic encryption scheme and let a be an unknown element of $\{m_0, m_1\}$. Then, the ciphertext for negation of a is given by*

$$E[\overline{a}] = E[a] \times E[m_1].$$

Proof. The proof is straightforward using Lemma 2.1. □

Lemma 3.2 (conjunction and disjunction) *Let $E[a]$ be a ciphertext of a homomorphic encryption scheme where a is an unknown plaintext in $\{m_0, m_1\}$. Let us assume b is a known (private) plaintext in $\{m_0, m_1\}$. Then, the ciphertext of conjunction (AND) of a and b is obtained without learning a as follows:*

$$E[ab] = \begin{cases} E[m_0] & \text{if } b = m_0, \\ E[a] & \text{if } b = m_1, \end{cases} \tag{4}$$

and the disjunction (OR) is

$$E[a \vee b] = \begin{cases} E[m_1] & \text{if } b = m_1, \\ E[a] & \text{if } b = m_0. \end{cases} \tag{5}$$

Proof. When $b = m_0$, the ciphertext $E[ab] = E[m_0]$ regardless of a. When $b = m_1$, the ciphertext $E[ab]$ is of m_1 only if $a = m_1$. The proof of disjunction is shown similarly. □

When the output ciphertext is required to be indistinguishable to anyone, including the owner of a, $E[a]$ in Eq. (4) can be re-encrypted using a random ciphertext $E[1]$ as $E[a]' = E[a] \times E[1]$.

Note that the plaintext a is not necessary to obtain $E[ab]$ and therefore conjunction consisting of multiple literals is feasible in the same manner. For example, from $E[a_1a_2a_3]$ and plaintext b, we can have $E[a_1a_2a_3b]$.

Lemma 3.3 *Let $E[a]$, $E[b]$ and $E[ab]$ be ciphertexts of secret plaintext in M. Then, the ciphertext of disjunction (OR) of a and b is*

$$E[a \vee b] = E[a] \times E[b] \times E[ab].$$

Proof. The proof is shown immediately from Lemma 2.1. □

3.2 Basic Protocol

We consider an oblivious party $\mathcal{C}$ which has a set of ciphertexts of internal state and updates the state in a publicly verifiable manner. Before we consider a specific function that requires less communication cost, we more generally show functional completeness of ciphertext computation at an oblivious party $\mathcal{C}$.

Let S_i be a set of ciphertext which contains internal private state in $\mathcal{C}$. The i-th state set S_i is of the form $\{s_1, s_2, \ldots, s_{L_i}\}$, defined by $s_1 = E[a_1], s_2 = E[a_2], \ldots, s_{L_i} = E[a_1 a_2 \cdots a_i]$, and $L_i = |S_i| = 2^i$, where a plaintext a_i is either m_0 or m_1. In particular, let S_0 be $\{m_1\}$.

1. Computer $\mathcal{C}$ sends to player P_i a current status $S_{i-1} = \{m_1, s_1, s_2, \ldots, s_{L_{i-1}}\}$.
2. For every element in S_{i-1}, player P_i use Eq. (4) in the conjunction protocol to compute the conjunctions of his private input a_i, and sends back to $\mathcal{C}$ the result
$$A_i = \{E[a_i], E[a_1 a_i], E[a_2 a_i], \ldots, E[a_1 a_2 \cdots a_{i-1} a_i]\}.$$
(Note that the constant m_1 in S_0 yields $E[m_1 a_i] = E[a_i]$ in A_i for every i.)
3. Computer $\mathcal{C}$ updates its state by
$$S_i = T(S_{i-1}, A_i).$$
4. Repeat 1 through 3 n times.
5. Computer $\mathcal{C}$ outputs $Y = D(S_n)$.

The transition algorithm T and the decoding algorithm D depend on the given boolean function f.

Theorem 3.1 (Functional Completeness) *For an arbitrary n-variable boolean function $f(x_1, \ldots, x_n)$, there exists a transition algorithm T and a decoding algorithm D such that*
$$Y = D(S_n) = E[f(a_1, \ldots, a_n)],$$
$$\begin{aligned} S_n &= T(S_{n-1}, A_n) = T(T(S_{n-2}, A_{n-1}), A_n) \\ &= \cdots = T(\cdots T(\{m_1\}, A_1) \cdots). \end{aligned}$$

Proof. By letting $T(S_{i-1}, A_i) = S_{i-1} \cup A_i$, we have the final state in which every element of power set of $\{a_1, a_2, \ldots, a_n\}$ is encrypted. Namely,
$$S_n = \{m_1, E[a_1], E[a_2], \ldots, E[a_1 a_2 \cdots a_n]\}.$$
From Lemma 2.3 and the homomorphism of encryption scheme, now we see that $\mathcal{C}$ has an arbitrary decoding algorithm D that produces the ciphertext of any given boolean function f. □

As an example of the decoding algorithm, let us consider a three-party majority function, which takes private boolean values a, b and c, and outputs whether more than two of them are 'true', or not. The basic protocol begins with $\mathcal{C}$ sending empty set to the first player who has a and sends back $A_1 = \{E[m_1 a]\} = \{E[a]\}$, which forms
$$S_1 = S_0 \cup A_1 = \{m_1, E[a]\}.$$
The second player computes the conjunction of $E[a]$ and her private b and sends to $\mathcal{C}$
$$A_2 = \{E[b], E[ab]\},$$
which leads to $S_2 = S_1 \cup A_2$. Similarly, the interaction with the third player provides the final state
$$S_3 = \{E[a], E[b], E[c], E[ab], E[ac], E[bc], E[abc]\}.$$
Finally, the computer $\mathcal{C}$ invokes the Reed-Muller representation for the objective majority function as follows
$$\begin{aligned} D(S_3) &= E[ab] \times E[ac] \times E[bc] \times E[abc] \\ &= E[ab \vee ac \vee bc]. \end{aligned}$$
Note that $\mathcal{C}$ learns nothing from the outcome of his decoding algorithm under the assumption of an indistinguishable encryption scheme, e.g., DDH.

In the above description, the basic protocol is the simplest in the sense that players have just one bit secret, which can be easily extended to k-bit secret in a natural way. In some target boolean function, the transition algorithm can also be replaced by one requiring less storage. In a later section, we will show a variation of the basic protocol in which a player has a k-bit secret and a more lightweight transition algorithm is used.

3.3 Verification Protocols

A malicious player may send an invalid input ciphertext to cheat other players or disrupt computation. To prevent players from violating the protocol without being detected, we require players to provide a proof of knowledge along with their ciphertexts. Although we have discussed a general homomorphic encryption scheme, we assume El Gamal encryption in this section.

Let $E[m] = (my^r, g^r) = (M, G)$ be a ciphertext encrypted with public key $y = g^x$. To prove $E[m]$ is valid ciphertext and m is either m_0 or m_1, a player who encrypts m shows a proof of knowledge of the form

$$PK\{(\alpha) : (M = m_0 y^\alpha \wedge G = g^\alpha) \vee (M = m_1 y^\alpha \wedge G = g^\alpha)\},$$

that is constructed based on the conjunctive and disjunctive proofs of knowledge in [CDS94].

The proof of knowledge is not sufficient because a malicious player can send a valid ciphertext as $E[ab]$ which is inconsistent with $E[a]$ and $E[b]$, e.g., claiming a forged ciphertext $E[m_0]$ to be $E[ab]$ but $a = m_1$. To prevent players from casting $E[ab] = (M_{ab}, G_{ab})$ inconsistent with $E[a] = (M_a, G_a), E[b] = (M_b, G_b)$ and $E[m_0] = (M_0, G_0)$, we can force them to send a proof of knowledge PK

$$\left\{ (\alpha, \beta) : \begin{pmatrix} M_a = m_0 y^\alpha \wedge \\ G_a = g^\alpha \wedge \\ M_{ab} = M_0 y^\beta \wedge \\ G_{ab} = g^\beta \end{pmatrix} \vee \begin{pmatrix} M_a = m_1 y^\alpha \wedge \\ G_a = g^\alpha \wedge \\ M_{ab} = M_b y^\beta \wedge \\ G_{ab} = g^\beta \end{pmatrix} \right\}.$$

Theorem 3.2 *Under an assumption of computational Zero-knowledge proof, any dishonest player ($\mathcal{C}, P_1, \ldots, P_n$) can not manipulate the result of computation in the basic protocol.*

4 Application to Auction Protocol

4.1 Oblivious Comparator

In this section, we first consider a standard (non-cryptographical) version of a k-bit integer comparator and then extend it to an oblivious comparator that compares input ciphertexts without decrypting.

Given k-bit integers a and b such that $a = a_{k-1}2^{k-1} + \cdots + a_1 2^1 + a_0 2^0$ and $b = b_{k-1}2^{k-1} + \cdots + b_1 2^1 + b_0 2^0$, an oblivious comparator $\mathcal{C}$ wishes to have c such that $c = a$ if $a > b$; otherwise $c = b$. Obviously, c is a k-bit integer represented as $c = c_{k-1}2^{k-1} + \cdots + c_1 2^1 + c_0 2^0$. The design of the comparator is simple if we have a one-bit comparator with two input bits a_i and b_i and three boolean variables, α, β and γ, such that α is true only if $a > b$, β is true only if $a < b$, and γ is true only if $a \neq b$. At first, all the variables are false.

For example, we show the logic formulas of the comparator when $k = 3$ as follows:

$$\begin{aligned}
c_2 &= a_2 \vee b_2, \\
\alpha_2 &= a_2 \overline{b_2}, \\
\beta_2 &= \overline{a_2} b_2, \\
\gamma_2 &= \alpha_2 \vee \beta_2, \\
c_1 &= \gamma_2(\alpha_2 a_1 \vee \beta_2 b_1) \vee \overline{\gamma_1}(a_1 \vee b_1), \\
\alpha_1 &= \alpha_2 \vee a_1 \overline{b_1}, \\
\beta_1 &= \beta_2 \vee \overline{a_1} b_1, \\
\gamma_1 &= \gamma_2 \vee \alpha_1 \vee \beta_1, \\
c_0 &= \gamma_1(\alpha_1 a_0 \vee \beta_1 b_0) \vee \overline{\gamma_1}(a_0 \vee b_0), \\
\alpha_0 &= \alpha_1 \vee a_0 \overline{b_0}, \\
\beta_0 &= \beta_1 \vee \overline{a_0} b_0, \\
\gamma_0 &= \gamma_1 \vee \alpha_0 \vee \beta_0.
\end{aligned}$$

After the evaluation of the above logic formulas, the greater integer in a and b is given by (c_2, c_1, c_0) and the boolean variables α_0, β_0 and γ_0 indicate whether $a > b$, $a < b$, and $a \neq b$, respectively. Note that γ_0 is false if and only if $a = b$.

As we have seen in the basic protocol, an arbitrary boolean formula can be represented in a Reed-Muller expression. For example, the boolean variable $\gamma_2 = \alpha_2 \vee \beta_2 = a_2\overline{b_2} \vee \overline{a_2}b_2$ has the Reed-Muller expression $a_2 \oplus b_2$. One more complicated example of c_1 is provided by using identities in Lemma 2.1 as follows:

$$\begin{aligned} c_1 &= \gamma_2(\alpha_2 a_1 \vee \beta_2 b_1) \vee \overline{\gamma_2}(a_1 \vee b_1) \\ &= \gamma_2(\alpha_2 a_1 \vee \beta_2 b_1) \oplus \overline{\gamma_2}(a_1 \vee b_1) \\ &= (a_2 \oplus b_2)(\alpha_2 a_1 \vee \beta_2 b_1) \oplus (1 \oplus a_2 \oplus b_2)(a_1 \vee b_1) \\ &= a_1 \oplus b_1 \oplus a_1 a_2 \oplus a_1 b_1 \oplus a_2 b_1 \\ &\quad \oplus a_1 a_2 b_2 \oplus a_1 b_1 b_2 \oplus a_1 a_2 b_1 \oplus a_2 b_1 b_2 \oplus a_1 a_2 b_1 b_2. \end{aligned}$$

Now, let us suppose that a player with a_2, a_1 and a_0 performs the three rounds of the basic protocol at once, i.e., sends to $\mathcal{C}$

$$E[a_0], E[a_1], E[a_2], E[a_0 a_1], E[a_0 a_2], E[a_1 a_2], E[a_0 a_1 a_2].$$

The other player having b_0, b_1, b_2 also participates in the protocol and provides a batch of ciphertexts

$$E[b_0], E[b_1], E[b_2], E[b_0 a_0], E[b_0 a_1], \ldots, E[b_0 b_1 b_2 a_0 a_1 a_2].$$

As the result, $\mathcal{C}$ has many ciphertexts, sufficient to compose c_2, c_1, c_0 without decrypting.

The result of the comparison is still ciphertext, which can be used as an input ciphertext for subsequent comparison, i.e., $b'_0 = c_0, \ldots, b'_{k-1} = c_{k-1}$. Hence, the state transition algorithm for a k-bit comparator requires a constant size of internal state, which is independent of the number of integers to be examined. Indeed, we have

$$S_n = S_{n-1} = \cdots = S_1 = \{E[c_1], \ldots, E[c_k]\}.$$

4.2 Sealed-Bid Auction Protocol

A suitable application of the oblivious comparator is a sealed-bid auction, where n bidders having k-bit private bids try to determine the highest bid and the winner in a secure manner. The oblivious comparator allows us to provide a trustworthy auctioneer who has interaction with every bidder once and blindly compare bids. The auctioneer has no chance to manipulate the winning price because all the processing steps, including a transition algorithm T and a decoding algorithm D, are publicly verifiable in the sense that anyone can make sure of the validity of the internal state without any secret information. The secrecy of the winning price and the winner are assured until more than a threshold number of players who share the corresponding secret key agree to decrypt the resulting ciphertexts.

In addition to functions for oblivious comparison, we need one more computer to determine the winner. This is not difficult. We suppose that every player has an assigned identity, say ID, that is a k_2-bit integer such that $k_2 > \log n$. The identities are encrypted as well and then sent to $\mathcal{C}$ in conjunction with input ciphertext. Using the boolean variables α_0 and β_0 in the oblivious comparator protocol, the comparator, $\mathcal{C}$, updates an additional internal state $W = (w_1, w_2, \ldots, w_{k_2})$ as

$$w_i = \alpha_0 IDA_i \vee \beta_0 IDB_i,$$

for $i = 1, \ldots, k_2$, where IDA_i and IDB_i are the i-th digits of identities for player A (voter) and player B (internal state). When multiple voters are tied at the same highest price, W is never set and therefore the default value will appear in the decrypted result as the sign that a tie occured.

4.3 Performance

In the proposed protocol, the amount of bit that a bidder has to send is the sum of 2^k ciphertexts for encoding bid and $2^{k_2}(> \log n)$ ciphertexts for encoding identity, resulting the total of $(2^k + 2^{k_2})l$ where l is a size of ciphertext. By letting $l = 1024 \cdot 2$, $k = 10$ (that is, $2^k = 1024$ values are possible to assign as biding prices), we have the number of ciphertext for one bidder is 2048 and the expected time for sending is 0.2 second (in bandwidth of 1 Mbps).

5 Conclusion

We have proposed a protocol for Secure Function Evaluation (SFE) with ciphertext, in which n players with input ciphertexts collaborate to compute an output ciphertext of a known boolean function. The main result is that an arbitrary function evaluation is feasible without decrypting the input ciphertext in n rounds of communication with an oblivious computer. We have shown the oblivious comparator is suitable to construct a secure sealed-bid auction that satisfies privacy of bids, verifiability of bidders, accountability of auctioneers and efficient computation.

Bibliography

[Abe00] M. Abe, "Universally Verifiable Mix-Net with Verification Work Independent of the Number of Mix-Servers," *IEICE Trans. Fundamentals*, Vol. E83-A, No.7, July 2000.

[Beav00] D. Beaver, "Minimal-Latency Secure Function Evaluation", Proc. of *EUROCRYPT 2000*, LNCS 1807, pp.335-350, 2000.

[BGW87] M. Ben-Or, S. Goldwasser and A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation, STOC88, pp.1-10, 1988.

[BMR90] Donald Beaver, Silvio Micali, and Philip Rogaway, The round complexity of secure protocols, STOC, 1990, pp.503-513

[Cac99] C. Cachin, Efficient private bidding and auctions with an oblivious third party, ACM Conference on Computer and Communications Security, pp.120-127, 1999.

[CDM00] R. Cramer, I. Damgård and U. Maurer, "General Secure Multi-party Computation from any Linear Secret-Sharing Scheme", Proc. of *EUROCRYPT 2000*, LNCS 1807, pp.316-334, 2000.

[CDN01] R. Cramer, I. Damgård and J. B. Nielsen, "Multiparty Computation from Threshold Homomorphic Encryption," Proc. of *EUROCRYPT 2001*, LNCS 2045, pp.280-300, 2001.

[CDS94] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," Proc. of *CRYPTO '94*, pp.174-187, 1994.

[CGS97] R. Cramer, R. Gennaro and B. Schoenmakers, "A Secure and Optimally Efficient Multi-Authority Election Scheme," Proc. of *EUROCRYPT 1997*, 2001.

[CM99] J. Camenisch and M. Michels, "Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes," Proc. of *EUROCRYPT'99*, pp. 107-122, 1999.

[Cre00] Giovanni Di Crescenzo, "Private Selective Payment Protocols", in Proc. of Financial Cryptography 2000, pp.72-89, 2000.

[eBay] eBay, `http://www.ebay.com`.

[FR96] M. K. Franklin and M. K. Reiter, The design and implementation of a secure auction service, *IEEE Trans. on Software Engineering*, 22(5), pp. 302-312, 1996.

[GM84] S. Goldwasser, S. Micali, "Probabilistic Encryption", Journal of Computer and System Sciences, Vol.28, No.2, pp.270-299, 1984.

[GMW87] S. Goldwasser and S. Micali and A. Wigderson, "How to Play Any Mental Game, or a Completeness Theorem for Protocols with an Honest Majority," Proc. of *the Nienteenth Annual ACM STOC'87*, pp. 218-229, 1987.

[HTK98] M. Harkavy, J. D. Tygar, and H. Kikuchi, Electronic auction with private bids, In Third USENIX Workshop on Electronic Commerce Proceedings, pp.61-74, 1998.

[JJ00] M. Jakobsson and A. Juels, "Mix and Match: Secure Function Evaluation via Ciphertexts," Proc. of *ASIACRYPTO 2000*, LNCS 1967, pp. 162-177, 2000.

[KHT99] H. Kikuchi, M. Harkavy and J. D. Tygar, Multi-round anonymous auction, IEICE Trans. Inf.& Syst., E82-D(4), pp.769-777, 1999.

[KMO01] J. Katz, S. Myers and R. Ostrovsky, "Cryptographic Counters and Applications to Electronic Voting," Proc. of *EUROCRYPT 2001*, LNCS 2045, pp. 78-92, 2001.

[KM99] K. Kobayashi and H. Morita, Efficient sealed-bid auction with quantitative competition using one-way functions, In Technical Report of IEICE, ISEC99-30, pp.31-37, 1999.

[Kud98] M. Kudo, Secure electronic sealed-bid auction protocol with public key cryptography, *IEICE Trans. Fundamentals*, E81-A(1), pp.20-26, 1998.

[MS99] S. Miyazaki and K. Sakurai, A bulletin board-based auction system with protecting the bidder's strategy, in *Trans. of IPSJ*, 40 (8), pp.3229-3336, 1999 (in Japanese).

[NPS99] M. Naor, B. Pinkas and R. Sumner, Privacy preserving auctions and mechanism design, ACM Workshop on E-Commerce, 1999.

[Ped91] T. P. Pedersen, "A threshold cryptosystem without a trusted party," Proc. of *EUROCRYPTO '91*, pp.522-526, 1991.

[SA99] F. Stajano and R. Anderson, The cocaine auction protocol: on the power of anonymous broadcast, Proc. of Information Hiding Workshop 1999 (LNCS), 1999.

[Sak00] K. Sako, An auction protocol which hides bids of losers, In *Proc. of PKC'2000*, pp.422-432, 2000.

[Sas97] T. Sasao, "Easily Testable Realizations for Generalized Reed-Muller Expressions", *IEEE Trans. Computers*, Vol. 46, No. 6, pp.709-716, 1997.

[SS99] S G. Stubblebine and P. F. Syverson, Fair On-Line Auctions without Special Trusted Parties, in *Proc. of Financial Cryptography 1999*, LNCS 1648, 1999, pp.230-240.

[SYY99] T. Sander, A. Young, and M. Yung, "Non-Interactive CyptoComputing For NC1", In 40th IEEE Annual Symposium on Foundations of Computer Science, pp. 554-567, 1999.

[WI00] Y. Watanabe and H. Imai, Optimistic Sealed-Bid Auction Protocol, in *Proc. of SCIS2000*, B09, pp.1-8, 2000.

[WWW98] P. R. Wurman, W. E. Walsh and M. P. Wellman, Flexible Double Auctions for Electronic Commerce: Theory and Implementation, *Decision Support Systems*, 24, pp.17-27, 1998.

On the Symbolic Analysis of Low-Level Cryptographic Primitives: Modular Exponentiation and the Diffie-Hellman Protocol*

Michele Boreale
Dipartimento di Sistemi e Informatica
Università di Firenze
Via Lombroso 6/17, 50137 Firenze, Italy
boreale@dsi.unifi.it

Maria Grazia Buscemi
Dipartimento di Informatica
Università di Pisa
Via F. Buonarroti 2, 56100 Pisa, Italy
buscemi@di.unipi.it

Abstract

Automatic methods developed so far for analysis of security protocols only model a limited set of cryptographic primitives (often, only encryption and concatenation) and abstract from low-level features of cryptographic algorithms. This paper is an attempt towards closing this gap. We propose a symbolic technique and a decision method for analysis of protocols based on modular exponentiation, such as Diffie-Hellman key exchange. We introduce a protocol description language along with its semantics. Then, we propose a notion of symbolic execution and, based on it, a verification method. We prove that the method is sound and complete with respect to the language semantics.

1 Introduction

During the last decade, a lot of research effort has been directed towards automatic analysis of crypto-protocols. Tools based on finite-state methods ([Low96, CJM97, MMS97]) take advantage of a well established model-checking technology, and are very effective at finding bugs. Infinite-state approaches, based on a variety of symbolic techniques ([AL00, Bor01, CCM01, Hui99, MS01]), have emerged over the past few years. Implementations of these techniques (e.g. [Van02, BB01a, STA01]) are still at an early stage. However, symbolic methods seem to be very promising in two respects. First, at least when the number of sessions is bounded, they can accomplish a complete exploration of the protocol's state space: thus they provide *proofs or disproofs* of correctness - under Dolev-Yao-like [DY83] assumptions - even though the protocol's state space is infinite. Second, symbolic methods usually rely on representations of data which help to control very well state-explosion induced by communications.

The application of automatic methods has mostly been confined to protocols built around 'black-box' enciphering and hashing functions. In this paper, we take a step towards broadening the scope of symbolic techniques, so as to include a class of low-level cryptographic operations. In particular, building on the general framework proposed in [BB02b], we devise a complete analysis method for protocols that depend on modular exponentiation operations, like the Diffie-Hellman key-exchange [DH76]. We expect that our methodology may be adapted to other low-level primitives (like RSA encryption).

The Diffie-Hellman protocol is intended for exchange of a secret key over an insecure medium, without prior sharing of any secret. The protocol has two public parameters: a large prime p and a generator α for the multiplicative group $\mathcal{Z}_p^* = \{1, \ldots, p-1\}$. Assuming A and B want to establish a shared secret key, they proceed as follows. First, A generates a random private value $n_A \in \mathcal{Z}_p^*$ and B generates a random private value $n_B \in \mathcal{Z}_p^*$. Next, A and B exchange their public values ($\mathsf{exp}\,(x, y)$ denotes $x^y \bmod p$):

$$\begin{array}{lll} 1. & A \longrightarrow B: & \mathsf{exp}\,(\alpha, n_A) \\ 2. & B \longrightarrow A: & \mathsf{exp}\,(\alpha, n_B). \end{array}$$

*This work has been partially supported by EU within the FET - Global Computing initiative, projects MIKADO and PROFUNDIS and by MIUR project NAPOLI.

Finally, A computes the key as $K = \mathsf{exp}(\mathsf{exp}(\alpha, n_B), n_A) = \mathsf{exp}(\alpha, n_A \times n_B)$, and B computes the key as $K = \mathsf{exp}(\mathsf{exp}(\alpha, n_A), n_B) = \mathsf{exp}(\alpha, n_A \times n_B)$. Now A and B share the value K, and A can use it to, say, encrypt a secret datum d and send it to B:

$$3. \quad A \longrightarrow B: \quad \{d\}_K.$$

The protocol's security depends on the difficulty of the discrete logarithm problem: it is computationally infeasible to compute y if only x and $\mathsf{exp}(x, y)$ are known.

When defining a model for low-level protocols of this sort, one is faced with two conflicting requirements. On one hand, one should be accurate in accounting for the operations involved in the protocol (exponentiation, product) and their 'relevant' algebraic laws; even operations that are not explicitly mentioned in protocols, but that are considered feasible (like taking the k^{th} root modulo a prime, and division) must be accounted for, because an adversary could in principle take advantage of them.

On the other hand, one must be careful in keeping the model effectively analysable. In this respect, recent undecidability results on related problems of equational unification [KNW03] indicate that some degree of abstraction is unavoidable. The limitations of our model are discussed in Section 2. Technically, we simplify the model by avoiding explicit commutativity laws and by keeping a free algebra model and ordinary unification. In fact, we 'promote' commutativity to non-determinism. As an example, upon evaluation of the expression $\mathsf{exp}(\mathsf{exp}(\alpha, n), m)$, an attacker will non-deterministically produce $\mathsf{exp}(\alpha, m \times n)$ or $\mathsf{exp}(\alpha, n \times m)$. The intuition is that if there is some action that depends on these two terms being equal modulo $\times$-commutativity, then there is an execution trace of the protocol where this action will take place. This seems reasonable in view of the fact that *we only consider safety properties* (i.e., 'no bad action ever takes place').

Here is a more precise description of our work. In Section 2, parallelling [BB02b], we introduce a syntax for expressions (including $\mathsf{exp}(\cdot, \cdot)$ and related operations), along with a notion of evaluation. Based on this, we present a small protocol description language akin to the applied pi [AF01] and its (concrete) semantics. The latter assumes a Dolev-Yao adversary and is therefore infinitary. In Section 3, we introduce a finitary symbolic semantics, which relies on a form of narrowing strategy, and discuss its relationship with the concrete semantics. A verification method based on the symbolic semantics is presented in Section 4: the main result is Theorem 4.3, which asserts the correctness and completeness of the method with respect to the concrete model. It is remarkable that the presence of the modular root operation plays a crucial role in the completeness proof. In Section 5, we illustrate how the method discovers the well known man-in-the-middle attack on the Diffie-Hellman protocol. A discussion on further research is in Section 6. A separate Appendix contains some most technical details. Due to space limitations, proofs are sketched or omitted but will appear in a forthcoming full version of the paper.

Related Work Very recent work by Millen and Shmatikov [MS03] shows how to reduce the symbolic analysis problem in the presence of modular exponentiation and multiplication plus encryption to the solution of quadratic Diophantine equations; decidability, however, remains an open issue. Pereira and Quisquater first [PQ01] proposed a technique for analysing group Diffie-Hellman protocols in the presence of an attacker with restricted capabilities (e.g. no symmetric encryption), though not facing the issue of decidability. Blanchet's model [BLa01] abstracts away from operations like inverse, root extraction and random number generation. The resulting method may give rise to false attacks and may not terminate. Closely related to our problem is also protocol analysis in the presence of the xor operation, which has been recently proven to be decidable by Chevalier *et al.* [CKRT03] and, independently, by Comon-Lundh and Shmatikov [CS03].

2 The model

We recall the concept of *frame* from [BB02b], and tailor it to the case of modular exponentiation and multiplication.

Expressions and messages We consider two countable disjoint sets of *names* $m, n, \ldots \in \mathcal{N}$ and *variables* $x, y, \ldots \in \mathcal{V}$. The set $\mathcal{N}$ is in turn partitioned into a countable set of *local names* $a, b, \ldots \in \mathcal{LN}$ and a countable set of *environmental names* $\underline{a}, \underline{b}, \ldots \in \mathcal{EN}$: these two sets represent infinite supplies of fresh basic values (keys, random numbers,...) at disposal of processes and of the (hostile) environment, respectively. It is technically convenient also to consider a set of *marked variables* $\hat{x}, \hat{y}, \hat{z}, \ldots \in \widehat{\mathcal{V}}$, which will be used as place-holders for generic messages known to the environment. The set $\mathcal{N} \cup \mathcal{V} \cup \widehat{\mathcal{V}}$ is ranged over by $u, v, \ldots$. Given a signature Σ of function symbols $f, g, \ldots$, each coming with its arity (constants have arity 0), we denote by $\mathcal{E}_\Sigma$ the algebra of terms (or *expressions*) on $\mathcal{N} \cup \mathcal{V} \cup \Sigma$,

$$
\begin{array}{rl}
\textsc{Signature} & \Sigma \;=\; \{\, \alpha\,,\ \mathsf{unit}\,,\ 1\,,\ \{\cdot\}_{(\cdot)}\,,\ \mathsf{dec}_{(\cdot)}(\cdot)\,,\ \mathsf{exp}\,(\cdot,\cdot)\,,\ \mathsf{root}\,(\cdot,\cdot)\,, \\
& \qquad \cdot\times\cdot\,,\ \mathsf{mult}(\cdot,\cdot)\,,\ \mathsf{inv}(\cdot)\,,\ \mathsf{inv}'(\cdot)\,,\ (\cdot)^{-1}\,\} \\
\textsc{Factors} & f ::= u \ \mid\ u^{-1} \\
\textsc{Products} & F ::= 1 \ \mid\ f_1 \times \cdots \times f_k \\
\textsc{Keys} & K, H ::= f \ \mid\ \mathsf{exp}\,(\alpha, F) \\
\textsc{Messages} & M, N ::= F \ \mid\ K \ \mid\ \{M\}_K
\end{array}
$$

$$
\begin{array}{rll}
(\textsc{Dec}) & \mathsf{dec}_\eta(\{\zeta\}_\eta) \leadsto \zeta & \\
(\textsc{Mult}) & \mathsf{mult}(\zeta_1 \times \cdots \zeta_k, \zeta_{k+1} \times \cdots \times \zeta_n) \leadsto \zeta_{i_1} \times \cdots \times \zeta_{i_n} & 1 \le k < n \le l \\
(\textsc{Inv}_1) & \mathsf{inv}(\zeta_1 \times \cdots \times \zeta_n) \leadsto \mathsf{inv}'(\zeta_1) \times \cdots \times \mathsf{inv}'(\zeta_n) & n \le l \\
(\textsc{Inv}_2) & \mathsf{inv}'(\zeta^{-1}) \leadsto \zeta \qquad (\textsc{Inv}_3)\quad \mathsf{inv}'(\zeta) \leadsto \zeta^{-1} \qquad (\textsc{Inv}_4)\quad \mathsf{inv}'(\zeta) \times \zeta \leadsto \mathsf{unit} & \\
(\textsc{Unit}_1) & \mathsf{unit} \times \zeta \leadsto \zeta \qquad (\textsc{Unit}_2)\quad \mathsf{unit} \leadsto 1 & \\
(\textsc{Exp}) & \mathsf{exp}\,(\mathsf{exp}\,(\xi, \eta), \zeta) \leadsto \mathsf{exp}\,(\xi, \mathsf{mult}(\eta, \zeta)) & \\
(\textsc{Root}) & \mathsf{root}\,(\mathsf{exp}\,(\xi, \eta), \zeta) \leadsto \mathsf{exp}\,(\xi, \mathsf{mult}(\eta, \mathsf{inv}(\zeta))) & \\
(\textsc{Ctx}) & \dfrac{\zeta \leadsto \zeta'}{C[\zeta] \leadsto C[\zeta']} \qquad\qquad \textsc{Evaluation}\quad \zeta \downarrow \eta \quad \text{iff} \quad \zeta \leadsto^* \eta &
\end{array}
$$

Table 1: $\mathcal{F}_{DH}$, a frame for modular exponentiation

given by the grammar:

$$\zeta, \eta \quad ::= \quad u \ \mid\ f(\widetilde{\zeta})$$

where $\widetilde{\zeta}$ is a tuple of terms of the expected length. A *term context* $C[\cdot]$ is a term with a hole that can be filled with any expression ζ, thus yielding an expression $C[\zeta]$.

Definition 2.1 (a frame for exponentiation) *A* frame $\mathcal{F}$ *is a triple* $(\Sigma, \mathcal{M}, \downarrow)$*, where:*

- Σ *is a signature;*
- $\mathcal{M} \subseteq \mathcal{E}_\Sigma$ *is a set of* messages $M, N, \ldots$*;*
- $\downarrow \subseteq \mathcal{E}_\Sigma \times \mathcal{E}_\Sigma$ *is an* evaluation relation*.*

A frame for exponentiation and shared key cryptography, $\mathcal{F}_{DH} = (\Sigma, \mathcal{M}, \downarrow)$*, is presented in Table 1. We write* $\zeta \downarrow \eta$ *for* $(\zeta, \eta) \in \downarrow$ *and say that* ζ evaluates to η*.*

Besides shared-key encryption $\{\zeta\}_\eta$ and decryption $\mathsf{dec}_\eta(\zeta)$ (with η used as the key), the other symbols of Σ represent arithmetic operations modulo a fixed and public prime number, which is kept implicit. In particular, we have exponentiation $\mathsf{exp}\,(\zeta, \eta)$, root extraction $\mathsf{root}\,(\zeta, \eta)$, a constant α that represents a public generator and two constants for multiplicative unit (unit, 1), two distinct symbols for the product $\mathsf{mult}(\zeta, \eta)$ and its result $\zeta \times \eta$, three symbols, $\mathsf{inv}(\zeta)$, $\mathsf{inv}'(\zeta)$ and ζ^{-1}, representing the multiplicative inverse operation. The reason for using different symbols for the same operation is discussed below. All the underlying operations are computationally feasible[1]. A message is either a product of up to l values, for a fixed l, or a key or a message encrypted under a key. A key can be an atomic object, or an exponential with basis α and a product as exponent ($\mathsf{exp}\,(\alpha, F)$).

Evaluation ($\downarrow$) is the reflexive and transitive closure of an auxiliary relation $\leadsto$, as presented in Table 1. There, we use $\zeta_1 \times \zeta_2 \times \cdots \times \zeta_n$ as a shorthand for $\zeta_1 \times (\zeta_2 \times \cdots \times \zeta_n)$, while $(i_1, \ldots, i_n)$ is any permutation of $(1, \ldots, n)$. The relation $\leadsto$ is terminating, but not confluent. In fact, the non-determinism of $\leadsto$ is intended to model the commutativity and the associativity of the product operation, as reflected in the rule (MULT). Also note rule (ROOT): in modular arithmetic, taking the k^{th} root amounts to raising to the $(k^{-1} \bmod p-1)^{th}$ power. The adoption of distinct symbols for

[1] Note that the inverse of k modulo n is defined only if $\gcd(k, n) = 1$ (see [MOV97]). Our model abstracts away from this condition. Another abstraction we make is that just one operation is used to model both inverse mod p and inverse mod $p-1$ (the latter operation arises only inside exponents).

$$
\begin{array}{rcll}
A, B & ::= & & \\
& & \mathbf{0} & \text{(null)} \\
& | & \mathsf{a}(x).\, A & \text{(input)} \\
& | & \overline{\mathsf{a}}\langle\zeta\rangle.\, A & \text{(output)} \\
& | & \mathsf{let}\, x = \zeta \,\mathsf{in}\, A & \text{(evaluation)} \\
& | & [\zeta = \eta]A & \text{(matching)} \\
& | & A \parallel B & \text{(parallel composition)} \\
& | & (\mathsf{new}\, a)\, A & \text{(restriction)}
\end{array}
$$

Table 2: Syntax of agents

product (mult and $\times$), inverse (inv, inv' and $()^{-1}$), and unit (unit and 1), and the form of the rules, ensure termination of both $\rightsquigarrow$ and of the induced narrowing relation, to be introduced in the next section.

The choice of the above message and rule formats corresponds to imposing the following restrictions on the attacker and on the honest participants:

1. there is a fixed upper bound (l) on the number of factors;
2. product and inverse operations cannot be applied to exponentials and to encrypted terms;
3. exponentiation starts from the basis α, and exponents can only be products.

More accurately, starting from a term obeying the above conditions, an attacker is capable of 'deducing' all - though not necessarily only - AC variants of the message represented by the term, in a sense made precise below. Terms not obeying these restrictions are just not guaranteed to produce any message. Restriction (1) might be relaxed at the cost of introducing a class of mult_l operations, for each $l \geq 0$, but for simplicity we shall stick to the above model in this paper.

Below, we define a deduction relation which expresses how the environment can generate new messages starting from an initial set of messages S. Note that environmental names and marked variables are treated as terms known to the environment. We denote by $\mathcal{P}_f(X)$ the set of finite subsets of X.

Definition 2.2 (deduction relation) *For $S \subseteq \mathcal{M}$, the set $\mathcal{H}(S)$ is inductively defined by the following rules:*

$$
\begin{array}{rcl}
\mathcal{H}^0(S) & = & S \cup \mathcal{EN} \cup \widehat{\mathcal{V}} \\
\mathcal{H}^{i+1}(S) & = & \mathcal{H}^i(S) \cup \{ f(\widetilde{\zeta}) : f \in \Sigma, \widetilde{\zeta} \subseteq \mathcal{H}^i(S) \} \\
\mathcal{H}(S) & = & \bigcup_{i \geq 0} \mathcal{H}^i(S).
\end{array}
$$

The deduction relation $\vdash \subseteq \mathcal{P}_f(\mathcal{M}) \times \mathcal{M}$ *is defined by:*

$$S \vdash M \quad \textit{iff} \quad \exists\, \zeta \in \mathcal{H}(S) : \zeta \downarrow M.$$

Example 2.3 *Consider a set $S = \{ n_A, \mathsf{exp}\,(\alpha, n_B) \}$. Then, $S \vdash \mathsf{exp}\,(\alpha, n_A \times n_B)$ and $S \vdash \mathsf{exp}\,(\alpha, n_B \times n_A)$. A further example involves the use of the root extraction. Consider a set $S = \{ \{m\}_{\mathsf{exp}\,(\alpha, k \times l)}, \mathsf{exp}\,(\alpha, k \times h), h, l \}$. Then, $S \vdash m$ since there exists $\zeta \in \mathcal{H}(S)$, $\zeta = \mathsf{dec}_\eta(\{m\}_{\mathsf{exp}\,(\alpha, k \times l)})$, with $\eta = \mathsf{exp}\,(\mathsf{root}\,(\mathsf{exp}\,(\alpha, k \times h), h), l)$, s.t. $\zeta \downarrow m$.*

Processes We consider a variant of the applied pi [AF01]. The syntax of *agents* is presented in Table 2.

Terms ζ and η range over $\mathcal{E}_\Sigma$. We consider a set $\mathcal{L}$ of *labels* which is ranged over by $\mathsf{a}, \mathsf{b}, \ldots$. A unique public channel is assumed, thus input and output labels ($\mathsf{a}, \mathsf{b}, \ldots$) are simply 'tags' attached to process actions for ease of reference. The only construct (let) for evaluating expressions replaces the ad-hoc constructs found in the spi-calculus [AG99] for encryption, decryption and other cryptographic operations. The construct $(\mathsf{new}\, a)\, A$ denotes the creation of a new name a, which is private to A. Note that the occurrences of variable x are bound in input and let operators.

Given the presence of binders for variables, notions of *free variables*, $\mathrm{v}(A) \subseteq \mathcal{V}$, *substitution* $[\zeta/u]$ for any ζ and u, and *alpha-equivalence* arise as expected. We shall identify alpha-equivalent agents. We denote by $\mathrm{en}(A)$ the set of environmental names occurring in A. An agent A is said to be *closed* or a *process* if $\mathrm{v}(A) = \emptyset$; the set of processes $\mathcal{P}$ is ranged over by $P, Q, \ldots$.

$$
\begin{array}{llcll}
(\textsc{Inp}) & \langle s,\ \mathsf{a}(x).\,P\rangle & \longrightarrow & \langle s\cdot \mathsf{a}\langle M\rangle,\ P[^{M}\!/\!_{x}]\rangle & s \vdash M \\
(\textsc{Out}) & \langle s,\ \overline{\mathsf{a}}\langle \zeta\rangle.\,P\rangle & \longrightarrow & \langle s\cdot \overline{\mathsf{a}}\langle M\rangle,\ P\rangle & \zeta \downarrow M \\
(\textsc{Let}) & \langle s,\ \mathsf{let}\, y=\zeta\,\mathsf{in}\, P\rangle & \longrightarrow & \langle s,\ P[^{\eta}\!/\!_{y}]\rangle & \zeta \downarrow \eta \\
(\textsc{Match}) & \langle s,\ [\zeta=\eta]P\rangle & \longrightarrow & \langle s,\ P\rangle & \zeta \downarrow \xi,\ \eta \downarrow \xi \\
(\textsc{New}) & \langle s,\ (\mathsf{new}\, a)\, P\rangle & \longrightarrow & \langle s,\ P\rangle & a = \mathrm{new}_{\mathcal{LN}}(V)
\end{array}
$$

$$
(\textsc{Par})\qquad \frac{\langle s,\ P\rangle \longrightarrow \langle s',\ P'\rangle}{\langle s,\ P \parallel Q\rangle \longrightarrow \langle s',\ P' \parallel Q\rangle}
$$

plus symmetric version of (PAR).
In (NEW), V is the set of free names in the source configuration.

Table 3: Rules for the concrete transition relation ($\longrightarrow$)

Example 2.4 (the Diffie-Hellman key exchange) *The process P below is a formalisation of the Diffie-Hellman protocol presented in the introduction. For simplicity, we just describe a one-session version of the protocol.*

$$
\begin{array}{lcl}
A & = & (\mathsf{new}\, n_A)\, \overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle.\, \mathsf{a2}(x).\, \mathsf{let}\, z=\mathsf{exp}\,(x, n_A)\,\mathsf{in}\, \overline{\mathsf{a3}}\langle\{d\}_z\rangle.\, \mathbf{0} \\
B & = & (\mathsf{new}\, n_B)\, \mathsf{b1}(y).\, \overline{\mathsf{b2}}\langle \mathsf{exp}\,(\alpha, n_B)\rangle.\, \mathsf{let}\, w=\mathsf{exp}\,(y, n_B)\,\mathsf{in}\, \mathsf{b3}(t). \\
 & & \mathsf{let}\, t'=\mathsf{dec}_w(t)\,\mathsf{in}\, B' \\
P & = & A \,||\, B.
\end{array}
$$

Here B' is a continuation of B after the reception of the encrypted datum d.

Operational Semantics The semantics of the calculus is given in terms of a transition relation $\longrightarrow$, which will sometimes be referred to as 'concrete' (as opposed to the 'symbolic' one we shall introduce later on). We model the state of the system as a pair $\langle s,\ P\rangle$, where s records the current environment's knowledge (i.e., the sequence of messages the environment has 'seen' on the network up to a given moment) and P is a process term. Formally, an *action* is a term of the form $\mathsf{a}\langle M\rangle$ (*input* action) or $\overline{\mathsf{a}}\langle M\rangle$ (*output* action), for a a label and M a message. The set of actions Act is ranged over by $\alpha, \beta, \ldots$, while the set Act^* is ranged over by $s, s', \ldots$. String concatenation is written '$\cdot$'. We denote by $\mathrm{act}(s)$ and $\mathrm{msg}(s)$ the set of actions and messages, respectively, appearing in s, and write '$s \vdash M$' for $\mathrm{msg}(s) \vdash M$.

Below we define *traces*, that is, sequences of actions that may result from the interaction between a process and its environment. In traces, each message received by a process (input message) can be synthesised from the knowledge the environment has previously acquired.

Definition 2.5 (traces and configurations) *A* trace *is a string $s \in Act^*$ such that for each s_1, s_2 and $\mathsf{a}\langle M\rangle$, if $s = s_1 \cdot \mathsf{a}\langle M\rangle \cdot s_2$ then $s_1 \vdash M$. A* configuration*, written as $\langle s,\ P\rangle$, is a pair consisting of a ground trace s and a process P. A configuration is* initial *if $\mathrm{en}(s, P) = \emptyset$. Configurations are ranged over by $\mathcal{C}, \mathcal{C}', \cdots$.*

The concrete transition relation on configurations is defined by the rules in Table 3. Each action taken by the process is recorded in the configuration's first component. Rule (INP) makes the transition relation infinitely-branching, as M ranges over the infinite set $\{M : s \vdash M\}$. In rule (OUT), ζ is evaluated and the action takes place only if the outcome is a message. By rule (LET), the evaluation of ζ replaces any free occurrence of y in P. In rule (NEW), a function $\mathrm{new}_{\mathcal{LN}}(V)$ is assumed that yields a local name $a \notin V$. No handshake communication is provided: all messages go through the environment (rule (PAR)). Note that, by (OUT), the non-determinism of $\leadsto$ is lifted to processes, and this is used to render commutativity and associativity of product. We are of course ignoring any 'efficiency' consideration, given that the model is infinite anyway (rule (INP)).

Properties We focus on correspondence assertions of the kind 'for every generated trace, whenever action β occurs in the trace, then action α must have occurred at some previous point in the trace'. Formally, given a configuration $\langle s,\ P\rangle$ and a trace s', we say that $\langle s,\ P\rangle$ *generates* s', written $\langle s,\ P\rangle \searrow s'$, if $\langle s,\ P\rangle \longrightarrow^* \langle s',\ P'\rangle$ for some P'. A substitution θ is a finite partial map from $\mathcal{V} \cup \widehat{\mathcal{V}}$ to the set of messages $\mathcal{M}$. For any object t (i.e. variable, message, process, trace, ...), we denote by $t\theta$ the result of simultaneously replacing each $x \in \mathrm{v}(t) \cap \mathrm{dom}(\theta)$ by $\theta(x)$. We let ρ range over ground substitutions, i.e. substitutions that map variables to ground messages.

Definition 2.6 (properties and satisfaction relation) *Let α and β be actions and s be a trace. We say that α* occurs prior to *β* in *s if whenever $s = s' \cdot \beta \cdot s''$ then $\alpha \in \mathrm{act}(s')$. For $\mathrm{v}(\alpha) \subseteq \mathrm{v}(\beta)$, we write $s \models \alpha \hookleftarrow \beta$, and say s* satisfies *$\alpha \hookleftarrow \beta$, if for each ground substitution ρ it holds that $\alpha\rho$ occurs prior to $\beta\rho$ in s. We say that a configuration $\mathcal{C}$* satisfies *$\alpha \hookleftarrow \beta$, and write $\mathcal{C} \models \alpha \hookleftarrow \beta$, if all traces generated by $\mathcal{C}$ satisfy $\alpha \hookleftarrow \beta$.*

Assertions $\alpha \hookleftarrow \beta$ can express interesting authentication (see [BB02b]) and secrecy properties. In particular, secrecy in the style of [AL00] can be set within our framework by assuming a conventional 'absurd' action $\perp$ that it is nowhere used in agent expressions. Thus, the formula $\perp \hookleftarrow \alpha$ means that action α should never take place.

Example 2.7 (Diffie-Hellman, continued) *The property that the protocol P in Example 2.4 should not leak the datum d can be expressed also by saying that the adversary will never be capable of synthesising d, without prior knowledge of it. This can be formalised by extending P with a 'guardian' process $\mathrm{g}(t).\,\mathbf{0}$ that at any time can pick up one message from the network and then stop: $S = P \parallel \mathrm{g}(t).\,\mathbf{0}$. Then we check whether this guardian can ever pick d from the network. This means checking whether the initial configuration $\mathcal{C}_{DH} = \langle \epsilon,\ S \rangle$ (ϵ = empty trace) satisfies the property $Secret(d) = \perp \hookleftarrow \mathrm{g}\langle d \rangle$, i.e. whether $\mathcal{C} \models Secret(d)$. Note that, by definition of deduction relation (Def. 2.2), $\epsilon \vdash \alpha$.*

3 Symbolic Semantics

We equip the frame $\mathcal{F}_{DH}$ with a *symbolic* evaluation relation ($\downarrow_s$), which is in agreement with its concrete counterpart ($\downarrow$). Intuitively, $\zeta \downarrow_\theta \eta$ means that ζ evaluates to η under all instances ρ of θ. The main advantage of the symbolic evaluation relation with respect to the concrete one is that infinitely many pairs (ζ, η) such that $\zeta \downarrow \eta$ can be represented by means of a single judgement $\zeta_0 \downarrow_\theta \eta_0$, for appropriate ζ_0, θ, η_0.

Formally, let us denote by $\mathrm{mgu}(t_1, t_2)$ a chosen *most general unifier* (mgu) of t_1 and t_2, that is, a unifier θ of t_1 and t_2 such that any other unifier is a composition of substitutions θ and θ', written $\theta\theta'$, for some θ'. Also, for t_1, t_1', t_2, t_2' terms, $\mathrm{mgu}(t_1 = t_1', t_2 = t_2')$ stands for $\theta\,\mathrm{mgu}(t_2\theta, t_2'\theta)$, where $\theta = \mathrm{mgu}(t_1, t_1')$, if such mgu's exist. The symbolic evaluation relation $\downarrow_s$ of $\mathcal{F}_{DH}$ is presented in Table 4: it is defined as the reflexive and transitive closure of the relation $\overset{\theta}{\leadsto}_s$.

Lemma 3.1 *Relation $\overset{\theta}{\leadsto}_s$ is image-finite and terminating. Hence, $\downarrow_s$ is image-finite.*

The notions of symbolic traces and symbolic configurations are defined below. The main difference from their concrete counterparts (Def. 2.5) is that no condition on input messages is required.

Definition 3.2 (symbolic traces and configurations) *A* symbolic trace *is a string $s \in Act^*$ such that: (a) $\mathrm{en}(s) = \emptyset$, and (b) for each s_1, s_2, α and x, if $s = s_1 \cdot \alpha \cdot s_2$ and $x \in \mathrm{v}(\alpha) - \mathrm{v}(s_1)$ then α is an input action. Symbolic traces are ranged over by $\sigma, \sigma', \ldots$. A* symbolic configuration*, written $\langle \sigma,\ A \rangle_{\mathsf{s}}$, is a pair composed by a symbolic trace σ and an agent A, such that $\mathrm{en}(A) = \emptyset$ and $\mathrm{v}(A) \subseteq \mathrm{v}(\sigma)$.*

The symbolic semantics is based on a symbolic transition relation $\longrightarrow_{\mathsf{s}}$, defined in Table 5. There, a function $\mathrm{new}_{\mathcal{V}}(\cdot)$ is assumed such that, for any given $V \subseteq_{\mathrm{fin}} \mathcal{V}$, $\mathrm{new}_{\mathcal{V}}(V)$ is a variable not in V. Moreover, $\mathcal{C} \xrightarrow{\theta}_{\mathsf{s}} \mathcal{C}'$ stands for $\mathcal{C} \longrightarrow_{\mathsf{s}} \mathcal{C}'$, where θ is the substitution applied to $\mathcal{C}'$ in the reduction step.

Note that, unlike the concrete semantics, input variables are *not* instantiated immediately (rule (INP$_{\mathsf{s}}$)). Rather, the input message is represented as a free variable and constraints on this variable are added as soon as they are needed, and recorded via mgu's. This may occur due to rules (OUT$_{\mathsf{s}}$), (LET$_{\mathsf{s}}$) and (MATCH$_{\mathsf{s}}$).

Example 3.3 *The simple example below shows how, after a first step, variable x is instantiated to name b by a (MATCH$_{\mathsf{s}}$)-reduction:*

$$\langle \epsilon,\ \mathsf{a}(x).\,[x = b]P \rangle_{\mathsf{s}} \ \longrightarrow_{\mathsf{s}}\ \langle \mathsf{a}\langle x \rangle,\ [x = b]P \rangle_{\mathsf{s}} \ \longrightarrow_{\mathsf{s}}\ \langle \mathsf{a}\langle b \rangle,\ P[^b/_x] \rangle_{\mathsf{s}}.$$

As a more elaborate example, let $P = \overline{\mathsf{a}}\langle k \rangle.\,\mathsf{a}(x).\,\mathsf{let}\, z = \mathsf{root}\,(x, k)\,\mathsf{in}\, P'$. After an output action and an input action, the symbolical evaluation of $\mathsf{root}\,(x, k)$ produces a global substitution $\theta = [^{\mathsf{exp}\,(x_1, k)}/_x]$ (x_1 fresh), to be applied to the whole configuration, and a local substitution $\theta' = [^{\mathsf{exp}\,(x_1, k)}/_z]$. I.e.

$$\langle \epsilon,\ P \rangle_{\mathsf{s}} \ \longrightarrow_{\mathsf{s}}^{*}\ \langle \sigma\theta,\ P'\theta\theta' \rangle_{\mathsf{s}}, \quad \textit{with } \sigma = \overline{\mathsf{a}}\langle k \rangle \cdot \mathsf{a}\langle x \rangle.$$

$$
\begin{array}{lll}
(\textsc{Dec}_\mathsf{s}) & \mathsf{dec}_\eta(\zeta) \overset{\theta}{\rightsquigarrow}_s x_1\,\theta & \theta = \mathrm{mgu}(\zeta = \{x_1\}_{x_2}, \eta = x_2) \\
(\textsc{Mult}_\mathsf{s}) & \mathsf{mult}(\zeta_1, \zeta_n) \overset{\theta}{\rightsquigarrow}_s (x_{i_1} \times \cdots \times x_{i_n})\,\theta & \left\{\begin{array}{l} 1 \le k < n \le l, \\ \theta = \mathrm{mgu}(\zeta_1 = x_1 \times \cdots \times x_k, \\ \zeta_2 = x_{k+1} \times \cdots \times x_n) \end{array}\right. \\
(\textsc{Inv1}_\mathsf{s}) & \mathsf{inv}(\zeta) \overset{\theta}{\rightsquigarrow}_s (\mathsf{inv}'(x_{i_1}) \times \cdots \times \mathsf{inv}'(x_{i_n}))\,\theta & \left\{\begin{array}{l} 1 \le n \le l, \\ \theta = \mathrm{mgu}(\zeta = x_1 \times \cdots \times x_n) \end{array}\right. \\
(\textsc{Inv2}_\mathsf{s}) & \mathsf{inv}'(\zeta) \overset{\theta}{\rightsquigarrow}_s x_1\,\theta & \theta = \mathrm{mgu}(\zeta, {x_1}^{-1}) \\
(\textsc{Inv3}_\mathsf{s}) & \mathsf{inv}'(\zeta) \overset{\epsilon}{\rightsquigarrow}_s \zeta^{-1} \qquad (\textsc{Inv4}_\mathsf{s}) \quad \mathsf{inv}'(\zeta) \times \eta \overset{\theta}{\rightsquigarrow}_s \mathsf{unit} & \theta = \mathrm{mgu}(\zeta, \eta) \\
(\textsc{Unit1}_\mathsf{s}) & \mathsf{unit} \times \zeta \overset{\epsilon}{\rightsquigarrow}_s \zeta \qquad (\textsc{Unit2}_\mathsf{s}) \quad \mathsf{unit} \overset{\epsilon}{\rightsquigarrow}_s 1 & \\
(\textsc{Exp1}_\mathsf{s}) & \mathsf{exp}\,(x, \zeta) \overset{\theta}{\rightsquigarrow}_s \mathsf{exp}\,(\alpha, \mathsf{mult}(x_1, \zeta)) & \theta = [\mathsf{exp}\,(\alpha, x_1)/x] \\
(\textsc{Exp2}_\mathsf{s}) & \mathsf{exp}\,(\mathsf{exp}\,(\xi, \eta), \zeta) \overset{\epsilon}{\rightsquigarrow}_s \mathsf{exp}\,(\xi, \mathsf{mult}(\eta, \zeta)) & \\
(\textsc{Root1}_\mathsf{s}) & \mathsf{root}\,(x, \zeta) \overset{\theta}{\rightsquigarrow}_s \mathsf{exp}\,(\xi, \mathsf{mult}(\eta, \mathsf{inv}(\zeta))) & \theta = [\mathsf{exp}\,(\alpha, x_1)/x] \\
(\textsc{Root2}_\mathsf{s}) & \mathsf{root}\,(\mathsf{exp}\,(\xi, \eta), \zeta) \overset{\epsilon}{\rightsquigarrow}_s \mathsf{exp}\,(\alpha, \mathsf{mult}(x_1, \mathsf{inv}(\zeta))) & \\
(\textsc{Ctx}_\mathsf{s}) & \dfrac{\zeta \overset{\theta}{\rightsquigarrow}_s \zeta'}{C[\zeta] \overset{\theta}{\rightsquigarrow}_s C\theta[\zeta']} &
\end{array}
$$

SYMBOLIC EVALUATION $\quad \zeta \downarrow_\theta \eta \quad$ iff $\quad \zeta \overset{\theta_1}{\rightsquigarrow}_s \cdots \overset{\theta_n}{\rightsquigarrow}_s \eta \quad$ and $\quad \theta = \theta_1 \cdots \theta_n$

Variables $x_1, \cdots, x_n$ are fresh according to some arbitrary but fixed rule.

Table 4: Symbolic Evaluation Relation ($\downarrow_s$) for $\mathcal{F}_{DH}$

$$
\begin{array}{lrcll}
(\textsc{Inp}_\mathsf{s}) & \langle \sigma,\ \mathsf{a}(x).\,A \rangle_\mathsf{s} & \longrightarrow_\mathsf{s} & \langle \sigma \cdot \mathsf{a}\langle x \rangle,\ A \rangle_\mathsf{s} & \\
(\textsc{Out}_\mathsf{s}) & \langle \sigma,\ \bar{\mathsf{a}}\langle \zeta \rangle.\,A \rangle_\mathsf{s} & \longrightarrow_\mathsf{s} & \langle \sigma\theta \cdot \bar{\mathsf{a}}\langle M \rangle,\ A\theta \rangle_\mathsf{s} & \zeta \downarrow_\theta M \\
(\textsc{Let}_\mathsf{s}) & \langle \sigma,\ \mathsf{let}\, x = \zeta \,\mathsf{in}\, A \rangle_\mathsf{s} & \longrightarrow_\mathsf{s} & \langle \sigma\theta,\ A\theta[\xi/x] \rangle_\mathsf{s} & \zeta \downarrow_\theta \xi \\
(\textsc{Match}_\mathsf{s}) & \langle \sigma,\ [\zeta = \eta]A \rangle_\mathsf{s} & \longrightarrow_\mathsf{s} & \langle \sigma\theta,\ A\theta \rangle_\mathsf{s} & \left\{\begin{array}{l} \zeta \downarrow_{\theta_1} \xi_1,\ \eta\theta_1 \downarrow_{\theta_2} \xi_2, \\ \theta_3 = \mathrm{mgu}(\xi_1\theta_2, \xi_2), \\ \theta = \theta_1\theta_2\theta_3 \end{array}\right. \\
(\textsc{New}_\mathsf{s}) & \langle \sigma,\ (\mathsf{new}\, a)\, A \rangle_\mathsf{s} & \longrightarrow_\mathsf{s} & \langle \sigma,\ A \rangle_\mathsf{s} & a = \mathrm{new}_{\mathcal{LN}}(V) \\
(\textsc{Par}_\mathsf{s}) & \multicolumn{3}{c}{\dfrac{\langle \sigma,\ A \rangle_\mathsf{s} \longrightarrow_\mathsf{s} \langle \sigma',\ A' \rangle_\mathsf{s}}{\langle \sigma,\ A \parallel B \rangle_\mathsf{s} \longrightarrow_\mathsf{s} \langle \sigma',\ A' \parallel B' \rangle_\mathsf{s}}} &
\end{array}
$$

plus symmetric version of ($\textsc{Par}_\mathsf{s}$). In the above rules it is assumed that:
(i) $x = \mathrm{new}_\mathcal{V}(V)$, being V the set of free variables in the source configuration;
(ii) $\mathrm{msg}(\sigma)\theta \subseteq \mathcal{M}$;
(iii) in rule ($\textsc{Par}_\mathsf{s}$), $B' = B\theta$ where $\langle \sigma,\ A \rangle_\mathsf{s} \xrightarrow{\theta}_\mathsf{s} \langle \sigma',\ A' \rangle_\mathsf{s}$.

Table 5: Rules for symbolic transition relation ($\longrightarrow_\mathsf{s}$)

Whenever $\langle\sigma,\ A\rangle_{\mathsf{s}} \longrightarrow_{\mathsf{s}}^{*} \langle\sigma',\ A'\rangle_{\mathsf{s}}$ for some A', we say that $\langle\sigma,\ A\rangle_{\mathsf{s}}$ *symbolically generates* σ', and write $\langle\sigma,\ A\rangle_{\mathsf{s}} \searrow_{\mathsf{s}} \sigma'$. The relation $\longrightarrow_{\mathsf{s}}$ is finitely-branching since $\downarrow_s$ is. Therefore, each configuration generates a finite number of symbolic traces. It is important to stress that many symbolic traces are in fact nonsense – sequences of actions that cannot be instantiated to any concrete trace. For instance, consider process $P = \mathsf{a}(y).\,\mathsf{let}\, x = \mathsf{dec}_k(y)\,\mathsf{in}\,\overline{\mathsf{a}}\langle x\rangle.\,\mathbf{0}$. The initial configuration $\langle\epsilon,\ P\rangle_{\mathsf{s}}$ symbolically generates the trace $\mathsf{a}\langle\{x_0\}_k\rangle\cdot\overline{\mathsf{a}}\langle x_0\rangle$, which is inconsistent, because the environment cannot generate the value k in $\{x_0\}_k$ (i.e. $\epsilon \not\vdash k$, hence $\epsilon \not\vdash \{x_0\}_k$). The problem of detecting these inconsistent traces, that might give rise to 'false positives' when checking protocol properties, will be faced in the next section.

Theorem 3.5 below establishes a correspondence between the concrete and the symbolic transition relations. It relies on the notion of consistency, defined below. Recall that marked variables are intended to carry messages known by the environment. For any $\hat{x}$ and any trace σ, we denote by $\sigma\backslash\hat{x}$ the longest prefix of σ not containing $\hat{x}$.

Definition 3.4 (consistency) *Let $\sigma \in Act^*$ and ρ be a ground substitution. We say that ρ* satisfies *σ if $\sigma\rho$ is a ground trace and, for each $\hat{x} \in \mathrm{v}(\sigma)$, it holds that $(\sigma\backslash\hat{x})\rho \vdash \rho(\hat{x})$. In this case we say that $\sigma\rho$ is a* solution *of σ, and that σ is* consistent*.*

Note that, because of the requirements on input messages, any trace (Def. 2.5) is obviously consistent: e.g., map all variables to some environmental names.

Theorem 3.5 (concrete vs. symbolic semantics) *$\mathcal{C}$ be an initial configuration and s be a ground trace. Then $\mathcal{C} \searrow s$ if and only if there exists σ such that $\mathcal{C} \searrow_{\mathsf{s}} \sigma$ and s is a solution of σ.*

4 A Verification Method

A crucial point of the method we present is checking consistency of symbolic traces. As mentioned in the previous section, a symbolic trace σ need not have solutions (ground instances that are traces). The next result allows us to check consistency.

Proposition 4.1 *Let σ be a symbolic trace. Then there exists a finite set of traces $\mathbf{Refinement}(\sigma)$, which are instances of σ and have the following property: for any s, s is a solution of σ if and only if s is a solution of some $\sigma' \in \mathbf{Refinement}(\sigma)$.*

Proposition 4.1 implies that σ is consistent if and only if $\mathbf{Refinement}(\sigma) \neq \emptyset$. Roughly, the set $\mathbf{Refinement}(\sigma)$ is computed by repeatedly unifying each input message in σ to terms that can be synthetized out of previous messages in σ. We refer to the Appendix for details. Here we shall content ourselves with giving a couple of examples; the second example shows the important role of the root extraction operation.

Example 4.2

1. *Consider $\sigma = \overline{\mathsf{c}}\langle\{m\}_k\rangle\cdot\overline{\mathsf{c}}\langle\{n\}_k\rangle\cdot\mathsf{c}\langle\{x\}_k\rangle$. Then $\mathbf{Refinement}(\sigma) = \{\ \sigma[^m\!/\!_x], \sigma[^n\!/\!_x]\ \}$.*
2. *Consider $\sigma' = \overline{\mathsf{c}}\langle h\rangle\cdot\mathsf{c}\langle x\rangle\cdot\overline{\mathsf{c}}\langle\mathsf{exp}\,(\alpha, k\times h)\rangle\cdot\overline{\mathsf{c}}\langle\{m\}_{\mathsf{exp}\,(\alpha,k\times x)}\rangle\cdot\mathsf{c}\langle m\rangle$. Clearly, σ' is consistent: e.g., map x to h. And, indeed, $\mathbf{Refinement}(\sigma') = \{\sigma''\}$ where $\sigma'' = \sigma'[^{\hat{x}}\!/\!_x]$. It is notable that the root extraction operation, though not mentioned in σ'', is essential to prove that σ'' is a trace. In fact, the environment is capable of learning m only by computing the key of the encrypted message as $\mathsf{exp}\,(\mathsf{root}\,(\mathsf{exp}\,(\alpha, k\times h), h), \hat{x})$.*

Let $\alpha \hookleftarrow \beta$ be a property and $\mathcal{C}$ an initial configuration. The verification method $\mathbf{M}(\mathcal{C}, \alpha \hookleftarrow \beta)$, presented in Table 6, checks whether $\mathcal{C} \models \alpha \hookleftarrow \beta$ or not. Moreover, if the property is not satisfied, $\mathbf{M}(\mathcal{C}, \alpha \hookleftarrow \beta)$ computes a trace violating the property, that is, an attack on $\mathcal{C}$.

To understand how the method works, it is convenient to consider the simple case $\alpha = \bot$, i.e. the verification of $\mathcal{C} \models \bot \hookleftarrow \beta$. This means verifying that in the *concrete* semantics, no instance of action β is ever executed starting from $\mathcal{C}$. By the correspondence between symbolic and concrete semantics (Theorem 3.5), this amounts to checking that for each σ symbolically generated by $\mathcal{C}$, no solution of σ contains an instance of β. The method proceeds as follows. First, one checks whether there is a mgu θ of γ and β, for every action γ of σ. If, for every σ, such a θ does not exist, or if it exists but $\sigma\theta$ is not consistent (this means that the check $\exists\,\sigma' \in \mathbf{Refinement}(\sigma\theta)$ at step 5 fails), then the property holds true, otherwise it does not, and the trace σ' violating the property is reported.

The correctness of the method in the general case is stated by Theorem 4.3 below. Note that the method always terminates, given that $\longrightarrow_{\mathsf{s}}$ is finite-branching and, hence, $\mathbf{Mod}_{\mathcal{C}}$ is finite.

$\mathbf{M}(\mathcal{C}, \alpha \hookleftarrow \beta)$
1. compute $\mathbf{Mod}_{\mathcal{C}} = \{\sigma \mid \mathcal{C} \searrow_{\mathsf{s}} \sigma\}$;
2. **foreach** $\sigma \in \mathbf{Mod}_{\mathcal{C}}$ **do**
3. **foreach** action γ in σ **do**
4. **if** $\exists\, \theta = \mathrm{mgu}(\beta, \gamma)$ **and**
5. $\exists\, \sigma' \in \mathbf{Refinement}(\sigma\theta)$ **where** $\sigma' = \sigma\theta\theta'$ **and**
6. $\alpha\theta\theta'$ does not occur prior to $\beta\theta\theta'$ in σ'
7. **then return**(No, σ');
8. **return**(Yes);

Table 6: The verification method

Theorem 4.3 (correctness and completeness) *Let $\mathcal{C}$ be an initial configuration and α and β be actions with* $\mathrm{v}(\alpha) \subseteq \mathrm{v}(\beta)$.

(1) *If $\mathbf{M}(\mathcal{C}, \alpha \hookleftarrow \beta)$ returns $(\mathsf{No},\ \sigma')$ then $\mathcal{C} \not\models \alpha \hookleftarrow \beta$. In particular, for any injective ground substitution $\rho : \mathrm{v}(\sigma') \to \mathcal{EN}$, $\mathcal{C} \searrow \sigma'\rho$ and $\sigma'\rho \not\models \alpha \hookleftarrow \beta$.*

(2) *If $\mathcal{C} \not\models \alpha \hookleftarrow \beta$ then $\mathbf{M}(\mathcal{C}, \alpha \hookleftarrow \beta)$ returns $(\mathsf{No},\ \sigma')$ and for any injective ground substitution $\rho : \mathrm{v}(\sigma') \to \mathcal{EN}$, $\mathcal{C} \searrow \sigma'\rho$ and $\sigma'\rho \not\models \alpha \hookleftarrow \beta$.*

Remark 4.4 *In practice, rather than generating the whole set of symbolic traces at once (step 1) and then checking the property, it is more convenient to work 'on-the-fly' and comparing every last symbolic action γ taken by the configuration against action β of the property $\alpha \hookleftarrow \beta$. The refinement procedure $\mathbf{Refinement}(\cdot)$ is invoked only when β and γ are unifiable.*

5 Example: Symbolic Analysis of the Diffie-Hellman Protocol

In this section we apply the verification method described in Section 4 to analyse the Diffie-Hellman protocol. We adopt the formalisation of the protocol presented in Examples 2.4 and 2.7.

The Diffie-Hellman protocol is subject to attacks from active adversaries. In terms of our model, discovering an attack on the protocol amounts to finding a ground trace s such that the initial configuration $\mathcal{C}_{DH} = \langle \epsilon,\ S\rangle \searrow s$ and $s \not\models \mathit{Secret}(d) = \bot \hookleftarrow \mathsf{g}\langle d\rangle$. And, indeed, such an s exists and is as follows:

$$\overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle \cdot \mathsf{a2}\langle \mathsf{exp}\,(\alpha, \underline{n}_I)\rangle \cdot \overline{\mathsf{a3}}\langle \{d\}_H\rangle \cdot \mathsf{g}\langle d\rangle,$$

where $\underline{n}_I$ is any environmental name and $K = \mathsf{exp}\,(\alpha, \underline{n}_I \times n_A)$.

Intuitively, the above trace corresponds to an attack in which the environment intercepts $\mathsf{exp}\,(\alpha, n_A)$, generates a private name $\underline{n}_I$ and handles $\mathsf{exp}\,(\alpha, \underline{n}_I)$ to A. Then, A computes $K = \mathsf{exp}\,(\alpha, \underline{n}_I \times n_A)$ and erroneously believes K is a shared key known by A and B. Finally, A sends over the network the secret datum d encrypted under the corrupted shared key K and, so, d is revealed to the environment.

Now, we have to show how this attack is discovered by our verification method. First, consider the following symbolic execution starting from $\mathcal{C}_{DH}$:

$$\begin{array}{ll}
\mathcal{C}_{DH} & \longrightarrow_{\mathsf{s}} \longrightarrow_{\mathsf{s}} \\
\longrightarrow_{\mathsf{s}} & \langle \overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle \cdot \mathsf{a2}\langle x\rangle, \mathsf{let}\ z = \mathsf{exp}\,(x, n_A)\ \mathsf{in}\ \overline{\mathsf{a3}\{d\}_z}\langle \mathbf{0}\rangle.\ \parallel B \parallel \mathsf{g}(t).\,\mathbf{0}\rangle_{\mathsf{s}} \\
\xrightarrow{\theta_0}_{\mathsf{s}} & \langle \overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle \cdot \mathsf{a2}\langle \mathsf{exp}\,(\alpha, x_0)\rangle, \overline{\mathsf{a3}}\langle \{d\}_K\rangle.\,\mathbf{0} \parallel B\theta_0\theta_1 \parallel \mathsf{g}(t).\,\mathbf{0}\rangle_{\mathsf{s}} \qquad (*) \\
\longrightarrow_{\mathsf{s}} & \langle \overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle \cdot \mathsf{a2}\langle \mathsf{exp}\,(\alpha, x_0)\rangle \cdot \overline{\mathsf{a3}}\langle \{d\}_K\rangle, \mathbf{0} \parallel B\theta_0\theta_1 \parallel \mathsf{g}(t).\,\mathbf{0}\rangle_{\mathsf{s}} \\
\longrightarrow_{\mathsf{s}} & \langle \overline{\mathsf{a1}}\langle \mathsf{exp}\,(\alpha, n_A)\rangle \cdot \mathsf{a2}\langle \mathsf{exp}\,(\alpha, x_0)\rangle \cdot \overline{\mathsf{a3}}\langle \{d\}_K\rangle \cdot \mathsf{g}\langle t\rangle, \mathbf{0} \parallel B\theta_0\theta_1 \parallel \mathbf{0}\rangle_{\mathsf{s}} \\
= & \langle \sigma,\ \mathbf{0} \parallel B\theta_0\theta_1 \parallel \mathbf{0}\rangle_{\mathsf{s}},
\end{array}$$

where $K = \mathsf{exp}\,(\alpha, x_0 \times n_A)$. In step $(*)$, rule $(\textsc{Let}_{\mathsf{s}})$ is applied, with $\mathsf{exp}\,(x, n_A) \downarrow_{\theta_0} \mathsf{exp}\,(\alpha, x_0 \times n_A)$, $\theta_0 = [{}^{\mathsf{exp}\,(\alpha, x_0)}/_{x}]$ (x_0 fresh), and $\theta_1 = [{}^{\mathsf{exp}\,(\alpha, x_0)}/_{z}]$.

Now, we execute the method step by step:

1. The symbolic model $\mathbf{Mod}_{\mathcal{C}_{DH}}$ is computed (in practice, symbolic traces are generated and checked 'on-the-fly').

2. The symbolic trace σ defined above is considered.

3,4. Action $\gamma = \mathsf{g}\langle t\rangle$ is found that unifies with $\beta = \mathsf{g}\langle d\rangle$, via $\theta = [d/t]$.

5. The set $\mathbf{Refinement}(\sigma\theta) = \{\sigma'\}$ is computed where $\sigma' = \sigma\theta\theta'$, and $\theta' = [\hat{x}_0/x_0]$. As stated by Theorem 4.1, σ' is a consistent trace. Note, in particular, that if we let $\sigma' = \sigma'' \cdot \mathsf{g}\langle d\rangle$, then $\sigma'' \vdash d$. Indeed, there exists $\zeta = \mathsf{dec}_\xi(\{d\}_\eta) \in \mathcal{H}(\sigma'')$, with $\eta = \mathsf{exp}\,(\alpha, \hat{x}_0 \times n_A)$, $\xi = \mathsf{exp}\,(\mathsf{exp}\,(\alpha, n_A), \hat{x}_0)$, and $\zeta \leadsto_s \leadsto_s \mathsf{dec}_\eta(\{d\}_\eta) \leadsto_s d$.

6. Action $\perp$ does not appear in σ', hence,

7. $(\mathsf{No},\ \sigma')$ is returned. Note that s is retrieved as $\sigma[\underline{n}_I/\hat{x}_0]$.

Of course, here we have proceeded by hand, but the symbolic traces generated by $\mathcal{C}_{DH}$ are of the order of hundreds, hence we expect that an implementation of the method would detect this attack in a fraction of second.

It is straightforward to extend the frame of Section 2 with operations such as pairing, public key encryption and decryption, hashing and digital signature; we refer the reader to [BB02b] for a detailed treatment of those operations. In particular, digital signature allows one to describe the authenticated version of the Diffie-Hellman protocol, where the public values exchanged are signed by the two partners, so that the man-in-the-middle attack is thwarted.

6 Conclusions and future work

We have presented a model for the analysis of protocols built around shared-key encryption and modular exponentiation. The model is less precise than others (notably [MS03]), its main limitation being a bound on the number of factors that may appear in any exponent. However, for such restricted model, we offer a decision method. The model and the method smoothly carry over when including other common enciphering, signing and hashing primitives. We also believe the method is effective in practice, because the symbolic model is compact, and the refinement procedure at its heart is only invoked on demand and on single symbolic traces. We are in the process of integrating the technique presented here into the STA analysis tool ([BB01a, STA01]).

Our technical development has been confined to multiplication and exponentiation, but the methodology presented suggests directions for extensions to other low-level primitives. It certainly seems promising for the case of RSA encryption, which is based on exponentiation too.

Acknowledgements. The authors wish to thank the anonymous referees for their helpful comments.

Bibliography

[AF01] M. Abadi, C. Fournet. Mobile Values, New Names, and Secure Communication. In *Conf. Rec. of POPL'01*, 2001.

[AG99] M. Abadi, A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1-70, 1999.

[AL00] R.M. Amadio, S. Lugiez. On the reachability problem in cryptographic protocols. In *Proc. of Concur'00*, LNCS 1877, Springer-Verlag, 2000. Full version: RR 3915, INRIA Sophia Antipolis.

[BLa01] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of 14th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2001.

[Bor01] M. Boreale. Symbolic Trace Analysis of Cryptographic Protocols. In *Proc. of ICALP'01*, LNCS 2076, Springer-Verlag, 2001.

[BB01a] M. Boreale, M. Buscemi. Experimenting with STA, a Tool for Automatic Analysis of Security Protocols. In *Proc. of SAC'02*, ACM Press, 2002.

[BB02b] M. Boreale and M. Buscemi. A Framework for the Analysis of Security Protocol. In *Proc. of CONCUR '02*, LNCS 2421. Springer-Verlag, 2002.

[CKRT03] Y. Chevalier, R. Kuesters, M. Rusinowitch, and M. Turuani. An NP Decision Procedure for Protocol Insecurity with Xor. In *Proc. of LICS '03*, IEEE Computer Society Press, 2003.

[CCM01] H. Comon, V. Cortier, J. Mitchell. Tree automata with one memory, set constraints and ping-pong protocols. In *Proc. of ICALP'01*, LNCS 2076, Springer-Verlag, 2001.

[CS03] H. Comon-Lundh and V. Shmatikov. Intruder Deductions, Constraint Solving and Insecurity Decision in Presence of Exclusive or. In *Proc. LICS '03*, IEEE Computer Society Press, 2003.

[DH76] W. Diffie, M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644-654, 1976.

[DY83] D. Dolev, A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29(2):198-208, 1983.

[Hui99] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. of Workshop on Formal Methods and Security Protocols*, Trento, 1999.

[KNW03] D. Kapur, P. Narendran, and L. Wang. An E-unification Algorithm for Analyzing Protocols that Use Modular Exponentiation. In *Proc. of RTA '03*, LNCS 2706, Springer-Verlag, 2003.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. of TACAS'96*, LNCS 1055, Springer-Verlag, 1996.

[Low97] G. Lowe. A Hierarchy of Authentication Specifications. In *Proc. of 10th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, 1997.

[CJM97] W. Marrero, E.M. Clarke, S. Jha. Model checking for security protocols. Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, 1997.

[MOV97] A Menezes, P.C. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, Boca Raton, 1997.

[MS01] J. Millen, V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. of 8th ACM Conference on Computer and Communication Security*, ACM Press, 2001.

[MS03] J. Millen and V. Shmatikov. Symbolic Protocol Analysis with Products and Diffie-Hellman Exponentiation. In *Proc. of 16th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, 2003.

[MMS97] J.C. Mitchell, M. Mitchell, U Stern. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proc. of Symp. Security and Privacy*, IEEE Computer Society Press, 1997.

[PQ01] O. Pereira and J.J Quisquater. A Security Analysis of the Cliques Protocols Suites. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, 2001.

[STA01] STA: a tool for trace analysis of cryptographic protocols. ML object code and examples. Available at `http://www.dsi.unifi.it/~boreale/tool.html`.

[Van02] V. Vanackère. The TRUST Protocol Analyser, Automatic and Efficient Verification of Cryptographic Protocols. In *Proc. of Verify '02*, 2002.

A Computing Refinement

Here we move within the general framework of [BB02b][2]. First, we introduce the concept of *basis*, which intuitively consists of the 'building blocks' of all messages deducible from a given σ. Given generic products F and G, we write $G \subset F$ to indicate that factors of G are strictly included in those of F.

[2] With a small change in terminology: in [BB02b], consistent symbolic traces are called 'solved forms', while $\mathbf{Refinement}(\cdot)$ is denoted by $\mathbf{SF}(\cdot)$

Definition A.1 (a basis for $\mathcal{F}_{DH}$) *For each symbolic trace σ:*

$$\begin{aligned}\mathbf{b}_{DH}(\sigma) = \{M \mid (\sigma \vdash M) \textit{ and } \big(& M \in \mathcal{LN} \cup \{\alpha, 1\} \\ & \textit{or } (M = F \textit{ and } \sigma \nvdash G, \forall G \subset F) \\ & \textit{or } (M = \mathsf{exp}(\alpha, F) \textit{ and } \sigma \nvdash G, \forall G \subset F)) \\ & \textit{or } (M = \{M\}_K \textit{ and } \sigma \nvdash K) \big) \}.\end{aligned}$$

In practice, for a given σ, the set $\mathbf{b}_{DH}(\sigma)$ can be effectively computed by an iterative procedure which repeatedly applies destructors (dec, root, mult, inv, inv') to messages in σ, until some fixed point is reached. This procedure always terminates and yields a finite set of messages.

Definition A.2 ($\mathbf{Refinement}(\sigma)$) *We let* refinement*, written* $\succ$*, be the least binary relation over symbolic traces given by the two rules below. In* (REF$_1$)*, it is assumed: that σ' is the longest prefix of σ which is a trace, that $\sigma = \sigma' \cdot \mathsf{a}\langle M\rangle \cdot \sigma''$, for some σ'', that $N, N' \notin \mathcal{V} \cup \widehat{\mathcal{V}}$, and that $\theta \neq \epsilon$.*

$$(\text{REF}_1)\ \frac{N' \in \mathbf{b}(\sigma') \quad M = C[N] \quad \theta = \mathrm{mgu}(N, N')}{\sigma \succ \sigma\theta\theta_0}$$

$$(\text{REF}_2)\ \frac{x \in \mathrm{v}(M)}{\sigma \succ \sigma[\hat{x}/x]}$$

where $\theta_0 = [\, x/\hat{x} \mid \hat{x} \in \mathrm{v}(\sigma) \textit{ and } |(\sigma\theta)\backslash\hat{x}| < |\sigma\backslash\hat{x}| \,]$.

For any symbolic trace σ, we let $\mathbf{Refinement}(\sigma) = \{\, \sigma' \mid \sigma \succ^{*} \sigma' \textit{ and } \sigma' \textit{ is a trace} \,\}$.

Rule (REF$_1$) implements the basic step of refinement: the subterm N of M gets unified, via θ, with an element of $\mathbf{b}(\sigma')$. By rule (REF$_2$) a variable x can become marked: this amounts to constraining the possible values of x to be messages known by the environment. (For technical reasons, marked variables sometimes need to be 'unmarked' back to plain variables, and this is achieved in (REF$_1$) via the renaming θ_0.)

Lemma A.3 $\mathbf{Refinement}(\sigma)$ *can be effectively computed, and is finite.*

Proof: It follows by two facts: (a) $\succ$ is an image-finite relation, and (b) infinite sequences of refinement steps cannot arise. As to the latter point, note that, since each (REF$_1$)-step eliminates at least one variable, any sequence of refinement steps can contain only finitely many (REF$_1$)-steps. After the last such step, rule (REF$_2$) can only be applied a finite number of times. □

Part V

Invited Talk

Language-Based Information Security

Andrei Sabelfeld
Cornell University
Ithaca, NY — USA
andrei@cs.cornell.edu

abstract

Modern computing systems are increasingly vulnerable to application-level attacks. These attacks are particularly dangerous because they circumvent the standard low-level protection mechanisms (such as OS-based monitors and access control). Furthermore, application-level attacks are often easy to create (or simply download and launch) - exactly because of their high-level nature. Because standard security low-level enforcement mechanisms offer only limited protection against application-level attacks, there is high demand for models of language-based security aimed at defending against threats at the programming-language and, hence, application level.

This talk concentrates on the preservation of information confidentiality by potentially malicious and/or buggy applications. Building on the technology of programming languages (such as programming-language semantics, type-based analysis, and program transformation) we develop a series of security policies and enforcement mechanisms for sequential, concurrent, and distributed programs that allow for modeling and statically analyzing information flow in a given program. The soundness of our security analyses guarantees the absence of insecure information flows. This means that if a program passes the analysis then it may not compromise confidentiality during the execution. We show that our approach is capable of detecting timing and probabilistic covert channels, i.e., the attacker is prevented from learning sensitive information by making timing and stochastic observations about secure programs.

Joint work with David Sands and Heiko Mantel.

Part VI

Language-Based Security

Implementation of Abstraction-carrying Code*

Songtao Xia James Hook

Department of Computer Science & Engineering
OGI School of Science & Engineering
Oregon Health & Science University
Portland, OR, USA
{sxia,hook}@cse.ogi.edu

Abstract

Abstraction-carrying code (ACC) provides a mechanism to certify temporal properties of mobile programs. This certification has many potential applications, including high confidence extension of an operating system kernel. The size of a traditional, proof-based certificate tends to expand drastically because of the state explosion problem. In ACC, we use an abstract interpretation of the mobile program as a certificate. A client receiving the code and the certificate will first validate the abstraction and then run a model checker to verify the temporal property. The size of a typical certificate for a non-trivial temporal property is usually significantly smaller than the size of the program.

We have developed ACCEPT, a prototype certifier for programs whose properties can be expressed as LTL formulas which do not use the next-operator. ACCEPT produces certificates for assembly programs; but it also assumes access to the higher level language source code as well. Several novel aspects of ACCEPT are: 1) the certifier produces a predicate abstraction which is used as a certificate; 2) the abstraction is valid for the source level program and the compiler maintains this validity for the assembly level program; 3) an index-typed assembly language SDTAL is adopted to encode the certificate; and 4) the abstraction is validated by type-checking the SDTAL program. This paper focuses on the representation of a predicate abstraction as type annotations on the assembly level program. The soundness of SDTAL's type system guarantees the soundness of the abstraction validation. We also report on our initial experience with ACCEPT.

1 Introduction

Proof-carrying code (PCC) techniques address the problem of trusting mobile (typically low-level) programs by verifying mathematical proofs for certain safety properties [Nec98]. Specifically, a server applies a safety policy to the program to generate verification conditions (VCs) and certifies the safety property with a checkable proof of the VCs. A client uses PCC to guarantee the safety of the program by generating his own set of VCs, and checking them against the supplied proof.

Early variants of PCC include TAL [MWCG99, MCGW98], ECC [Koz98], Certified Binary [SSTP02], FPCC [App01], TouchStone [Nec98] and others. They focus primarily on the verification of type-safety and memory-safety properties easily expressed in first-order logic. In this paper, we investigate the certification of more general temporal properties.

For example, safety policies, such as the correct use of APIs [BR01, HJM$^+$02], or liveness requirements, are expressible by temporal logic [MP92]. To enforce such a policy, we must be able to certify and verify temporal properties. An example of the potential benefits of the enhanced expressive power is that an OS kernel is able to start a service, developed by an untrusted party, and still be assured that the system will not be blocked. This is accomplished by verifying temporal properties of the driver. We can thus extend the OS kernel at run-time securely without running a watchdog.

Researchers have extended the PCC framework to address temporal properties. In these systems, a certificate is, as in PCC, a proof of the temporal property. Whether this proof is worked out using a theorem prover, as in the case of Temporal PCC (TPCC) [BL02], or translated from the computation of a model checker [TC01, Nam01, Nam03], the

*This research is partially sponsored by Department of Defense.

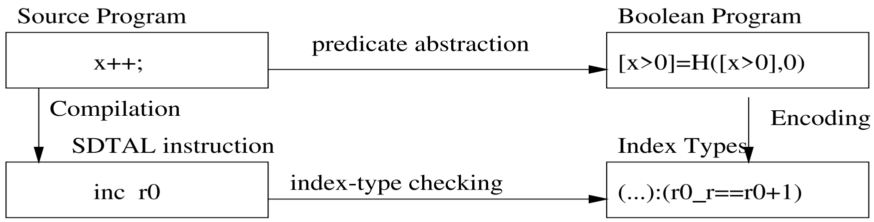


Figure 1: Relation between source program, assembly program, Boolean program and index-type

size of the proof tends to be large because of the state explosion problem. In some cases the proof (even for trivial properties) is several times larger than the program [BL02].

We are exploring Abstraction-carrying code (ACC)[XH03a], an alternative method for certifying and verifying general temporal properties. A server will send an abstract interpretation [CC77] of a mobile program as a certificate of the temporal property. A client first validates the abstraction and then model checks a temporal property (of the abstraction). If the property belongs to the class of properties preserved by the abstraction, then the property will hold on the mobile program. Often, we can model check an appropriately abstracted program much faster than the original program. A conceptually similar approach has been independently suggested by a group at Stony Brook [SRRS01].

This paper addresses the following two fundamental questions that, to the best of our knowledge, have not been previously addressed: 1) How do we compute an abstraction of an assembly program? 2) How do we compactly represent an efficiently verifiable abstraction? Our strategy is to use a multi-stage process. This is illustrated in Fig. 1. First, we adopt a refinement based [CGJ$^+$00, Kur94] predicate abstraction technique [BR01, Sai00, GS97, DDP99, BLO98] to compute a predicate abstraction of the source program. The result is a Boolean program. Second, we compile the source code into SDTAL (a Simplified Dependently Typed Assembly Language) program. Third, we encode the predicate abstraction as index-type annotations on the SDTAL program. A client type checks the SDTAL program to validate the predicate abstraction. From the type annotations the predicate abstraction can be recovered. To validate a temporal property, the recovered abstraction needs to be model checked. We have implemented this approach in a prototype toolkit called ACCEPT (ACC Evaluation Prototype Toolkit).

The primary advantage of our approach is that we are able to reduce the size of a certificate. Other researchers applying predicate abstraction techniques have been able to model check properties of engineering significance [BR01, HJM$^+$02, DDP99, HP00, CDH$^+$00]. Their experience shows that the size of predicate abstraction is in general small for these programs. This is further supported by our own initial observation. For example, ACCEPT is able to generate certificates significantly smaller than the assembly programs.

The paper is organized as follows. We develop a running example (a simplified Linux device driver code fragment) to introduce the techniques of ACC. We then provide some technical background: What kind of programs can be abstracted as boolean programs, and the use of indexed-types (Section 2). We show how to encode a Boolean program as index-type annotations. We introduce the dynamic and static semantics of SDTAL as well as a soundness result. These sections combine to answer the questions we raised above. Section 5 summarizes our experience with ACCEPT. Section 6 concludes.

2 The ACC framework

In Figure 2 a Java code fragment of a bounded buffer program with 2 buffers is displayed. It shows only one of two symmetric threads. Global variables $\texttt{buf}_1$ and $\texttt{buf}_2$ count the elements in the buffers. The capacity of both buffers is a constant `bound`. The thread shown constantly moves elements from one buffer to the other. The other thread does the opposite. One of the properties we expect to be valid is that whenever a buffer is full, then sometime in the future, one of the elements must eventually be removed. This property, which we name as FullToNonFull, can be expressed in a Linear Temporal Logic (LTL) term as

$\Box(\text{Full} \rightarrow \Diamond\text{NonFull})$ (FullToNonFull).

Model checking this property on the source program directly is relatively expensive due to the state explosion problem. Model checking can be thought of as observing what properties hold on all feasible control flows, i.e. all possible paths around loops and through branches, etc. If the number of paths is very large this becomes infeasible. A popular way to attack this problem is predicate abstraction, observing a program's behavior on a selected set of predicates rather than on its original variables. We use different predicates for different properties. For example, in the

```
// int buf1=bound, buf2=0;
// int bound= 50;
thread1 {
  run() {
  while (true){
    synchronized {
      while (buf1 <= 0)
       wait();
      buf1--;};
```

```
while (*)
  {assume (empty1); wait};
assume (! empty1);
  [full1 := H(0, full1
          || nonfull1),
   nonfull1 := H(full1
       || nonfull1, 0),
   empty1 := *;
  ]
```

```
    synchronized {
      while (buf2 >= bound)
       wait();
     buf2++;}
  }
  }
}
// thread2 symmetric to thread1
```

Figure 2: Bounded buffer example, with corresponding Boolean program fragment.

buffer example we may observe the six predicates below to answer the FullToNonFull property.

$$\begin{array}{rll} \{\text{Empty}_i: & \texttt{buf}_i = 0, & \text{where } i = 1, 2. \\ \text{Full}_i: & \texttt{buf}_i = \texttt{bound}, & \\ \text{NonFull}_i: & \texttt{buf}_i < \texttt{bound}\} & \end{array}$$

Choosing the correct predicates to observe is beyond the scope of this discussion [BR01, HJM$^+$02]. The Boolean program mimics the behavior of the original program on the selected predicates. The Boolean program is obtained by translating each statement in the concrete program (the program on the left in Figure 2) into a Boolean program which exclusively manipulates the boolean variables naming the predicates (e.g. Empty$_i$, Full$_i$, and NonFull$_i$). The result is the program on the right in Figure 2.

In a Boolean program, each statement in the original program is represented by an approximation which manipulates just the boolean variables. Branches in Boolean programs introduce some problems that need to be handled in a special manner. An if- statement in the concrete program is always translated into if-statement in the Boolean program with a test that is a non-deterministic choice, and with an `assume`-statement at the head of each branch. A similar translation is used for while statements. An `assume` statement is an annotation that allows the model checker to assume the condition when processing that branch. The two `assume`- statements in the right hand side of Fig.2 illustrates the translation. We need *two* different assume statements because the approximation of the negation of the if- condition is not the negation of the approximation of the if-condition.

An assignment in the source language, such as $\texttt{buf}_i$--, is usually translated into a group of simultaneous assignments in the Boolean program. We group such assignments in square brackets ([...]). The effects of $\texttt{buf}_i$-- on predicate Full$_i$, for example, is that, if Full$_i$ or NonFull$_i$ is true beforehand, then Full$_i$ will be false afterward; no combination of these predicates guarantees that Full$_i$ will be true. This is characterized by the H function. Intuitively, $H(e_1, e_2)$ returns true if e_1 is true, false if e_2 is true, or non-deterministic (*) (completely unknown), if neither e_1 nor e_2 is true.

Further treatment of predicate abstraction and Boolean programs can be found in [GS97, DDP99, BR01]. In the examples in this paper, we only use assignments, if-statements and while-statements (but not function calls, or recursion). This is sufficient for many applications, such as device drivers.

We also assume we have computed an adequate Boolean program for the source program. We can leverage existing tools [BR01, HJM$^+$02], which typically rely on the refinement based abstraction techniques to compute a Boolean program on which we are able to model check the property.

2.1 Problems with Constructing Abstractions for Assembly Code

Typically in PCC, a mobile program is compiled. When we evaluate a temporal property on an assembly program or its abstraction, we have to address a few additional important issues.

Abstraction. To characterize the same effect in a Boolean program abstracting an assembly program, we need more predicates than are needed to abstract a source level program. As an example, for an assignment $x = 2 * x + 1$ in Java and a predicate $\{x > 0\}$, we will have a Boolean assignment that states: if $x > 0$ beforehand, then it will be so afterward. But a corresponding assembly sequence, {mul x 2 r_0; add r_0 1 x} will need another predicate $\{r_0 > 0\}$ to capture the same information. Consequently, abstracting assembly code tends to generate a larger predicate set. Because computation of a Boolean program takes time exponential in the number of predicates, we choose to abstract source programs and to encode/transform the abstraction during compilation. This is feasible because during the compilation process we can group together several assembly level instructions into a single unit of abstraction (since we know the high-level source code they originated from).

Volatility. A variable is volatile if the compiler always emits a store instruction for it whenever it is modified. When this is the case the relationship between the high-level program and its assembly level program is more concrete. This simplifies matters because we can observe the atomic propositions in terms of the values stored in the memory, instead of their buffered copies in registers. We require that variables involved in the temporal property, such as $\texttt{buf}_i$ in our example, to be volatile. In this paper, we assume that the variables involved in the predicate set are also volatile. These variables are called observable variables.

Atomicity. A tightly related issue is that of atomicity. A hidden assumption in the predicate abstraction-based software model checking is that the source language element to be abstracted is atomic.

For example, in SLAM, a (pure) C statement that does not involve function calls, is considered atomic. Any (observable) side effect is hazardous. For example, suppose we first clear x (to 0) when compiling the assignment x=3. The assembly code will be:

$\{\texttt{mov}\ r_0\ 0; \texttt{store}\ r_0\ x\ ; \texttt{mov}\ r_0\ 3\ ; \texttt{store}\ r_0\ x\}$

If we are verifying the global property $x > 2$, a correct Boolean program of the compiled sequence is to assign *true* to a Boolean variable representing predicate $x > 2$. The boolean program model checks successfully. However, the property does not hold on the concrete assembly program. We mistakenly abstract away some states that violate the property. For this reason, we demand the compiled assembly level code to satisfy the following requirement.

Atomicity Requirement. A piece of assembly code compiled from an atomic source program element may have at most one store (to observable variables) in it.

Next, we describe how we compile a source program while preserving the abstraction. That is, the abstraction of the source program can be transformed to an abstraction of the assembly program. We also explain why we choose index-types to encode the Boolean program.

2.2 Abstraction-preserving Compilation

Intuitively, if an assembly level program, and the source program from which it is compiled share a common control flow diagram then the Boolean program produced for the source program may be reused. Below we describe a conservative compilation scheme that generates a Boolean program that not only is a valid abstraction of the assembly program but also model checks the temporal property. In the general case, the Boolean program does not have to be the same as the one for the source program, which is discussed at length in another paper [XH03b]. As a starting point, we will reuse the same Boolean program. We redraw Figure 1 in Figure 3; the dotted line represents the correspondence between the Boolean program and the assembly program.

Assume, for a concrete program C, we have a corresponding Boolean program B. To simplify presentation, C statements are all atomic assignments or if-statements. We also assume we will compile C into an untyped assembly program D. We assume that all (observable) variables in C have their counterparts in D. We use symbolic names for these assembly variables.

Ignoring recursion, we can organize both C and D as connected graphs of blocks. Edges in the blocks represent control flow. This is an informal presentation of a control flow automaton (CFA) [HJM+02]. An assignment in C is compiled to a block (called an assignment block) in D. We require an assignment block to have an unconditional jump at the end. An if-statement's conditional test is compiled to a block in D (called a test block), which must have a conditional jump. A conditional jump in the assembly language takes two labels. To conveniently handle assume statements, we insert a dummy block, which contains only an unconditional jump, before the D blocks corresponding to the branches. For example, an if-statement in C is compiled to five blocks. One computes the condition and decides which branch to take; two dummy blocks lead two blocks that correspond to the then-clause and else-clause, respectively.

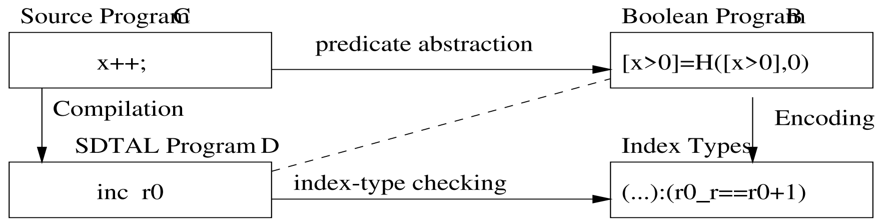


Figure 3: Relation between source program, assembly program, Boolean program

Figure 4: Snippet of the Bounded Buffer Example in CFA

Going back to the Boolean program, a D assignment corresponds to a simultaneous assignment group in B; a test block in D corresponds to non-deterministic choice in B; and a dummy block in D corresponds to an `assume` statement in B. Informally, this correspondence defines the abstraction relationship between a Boolean program and an assembly program.

A source program and an assembly program that share a CFA are illustrated in Figure 4.

Validation of B on D is thus to verify: 1) that an assignment block in D has the effect of its corresponding assignments in B, and 2) that the condition assumed by an assume statement in B always holds at the corresponding dummy block. Knowing the semantics of the assembly language, we can easily design an algorithm to validate the Boolean program and add this algorithm to the trust base (the set of programs trusted by a client) in a PCC system. But this program will be a special purpose one. In the context of mobile code verification, there are many artifacts to verify. For example, a client wants to verify memory safety properties besides the temporal property. Are we going to represent these artifacts all in a different way and write a different program to validate a different artifact?

Ideally, we wish to encode a Boolean program as type annotations in a typed assembly language [MWCG99, MCGW98, XH01] and validate the Boolean program through type checking. This way we can reuse the type checker of the assembly language. When multiple kinds of properties are to be verified, this approach saves us from inventing another representation and further increasing the trust base, as long as the properties are expressible in the logic handled by the type system.

Index-typed assembly languages [XH01] have previously verified standard PCC properties such as loop invariants or indexed function types. These properties are similar to the conditions required in validating a Boolean program. This fact inspires us to investigate if a Boolean program can be encoded as type annotations and can be validated through index-type checking. In the next subsection, we review the concept of index-types and present the type language of SDTAL.

2.3 Background: index-types in SDTAL

The type system of SDTAL derives from DTAL's [XH01], which applies index-types [Xi97, Zen98] to assembly languages. DTAL was designed for two distinct goals. One is to compile higher level index-typed languages, such as decaml. The other is to extend the type system of TAL [MWCG99], so that the resultant language can be used in a PCC framework. The type checking of DTAL can verify certain data-flow facts stated by the index-type annotations, such as loop invariants.

index variables	x, y	$::=$	x_i, v_i, x_{ri}
int constants	c		
expressions	e	$::=$	$x \mid c \mid e_1\ aop\ e_2$
arithmetic op	aop	$::=$	$+ \mid - \mid * \ldots$
indices	iv	$::=$	x
index propositions	P	$::=$	$e_1\ cop\ e_2 \mid \neg P \mid P_1\ lop P_2$
comparison op	cop	$::=$	$> \mid < \mid \ldots$
logic op	lop	$::=$	$\vee \mid \wedge \mid \Rightarrow \ldots$
index contexts	ϕ	$::=$	$\cdot \mid \phi, P$
unconstrained base type	t	$::=$	$\texttt{int(iv)} \mid (t_1, t_2)$
types	τ	$::=$	$t; P \mid (t_1; P_1 \to t_2; P_2)$
instructions	`ins`	$::=$	`a-ins` $R_1\ R_2\ R_3$
			`load` `var` R $\mid$ `store` R `var`
			$\mid$ `jmpcond` R_1 $\mid$ `mov` $R_1\ R_2$
ins sequence	I	$::=$	`ins`*
blocks	b	$::=$	I
labels	l		
annotations	`anno`	$::=$	$b : \tau \mid\ l : (\tau).b$
arithmetic ins	`a-ins`	$::=$	`add` $\mid$ `sub`...

Figure 5: Syntax of SDTAL

SDTAL simplifies DTAL to allow efficient encoding and verification of certain data-flow facts. SDTAL uses a more restricted language to describe the relationship among typable program entities, such as a register or a variable. SDTAL also has simpler type checking rules; and SDTAL adopts a more expressive decision procedure than the integer programming toolkit used by DTAL.

In Fig. 5, we list the syntax of SDTAL. Because recursion in general cannot be handled by model-checking, we assume all functions are inlined and omit the function calls. To simplify the presentation, we also eliminate the use of a stack, which has been treated, for example, by the technical report of Xi and Harper [XH01]. In SDTAL, arithmetic operations take operands from registers (or constants). Load and store are the only ways to access the memory.

In index-types, indices are expressions taken from one of the index domains to index a type term. For example, in index-type terms $\texttt{int}(5), \texttt{int}(x), \texttt{int}(2 * x)$, integer constant 5, index variable x and expression $2 * x$ are the indices, respectively. An index variable has a sort, which defines the index domain. For example, if x is an integer, then index-type $\texttt{int}(2 * x)$ is a type for even numbers. In SDTAL, all indices are index variables and the only index domain is integer. An index proposition constrains the index-type. For example, a type for even numbers can also be expressed as $\texttt{int}(x) : x == 2 * y$, where the index proposition is $x == 2 * y$. The index language in SDTAL contains uninterpreted function symbols, simple arithmetic theory, equality theory and propositional logic. In practice, this language is sufficient to express most relations involved in predicate abstraction.

An SDTAL program has two kinds of type annotations (τ in the syntax). One is called a type state, which specifies the condition held by a state. For example, a type state where variable m is greater than register r_0 is represented as $(int(x_0), int(x_1), ..., m...); m > x_0$. Generally, a type state is of the form $(R; P)$, where R is an implicit typing (unconstrained base type) of the machine state. R is a tuple

$$(\texttt{int}(x_0),\ \ \texttt{int}(x_1), ..., \texttt{int}(x_{n-1}), \texttt{int}(\texttt{var}_1), ...)$$

where the first n $\texttt{int}(x_i)$s are unconstrained base types for registers $\{r_0, ..., r_{n-1}\}$. A base type $\texttt{int}(x_i)$, where x_i is said to index the type $\texttt{int}$, associates the index variable x_i with an inhabitant of that type, namely r_i. The rest of the components in R are base types for program variables, indexed by index variables $\{\texttt{var}_1, ...\}$. In SDTAL, we fix the association between a value and its index variable, that is, r_0 is always typed by $\texttt{int}(x_0)$. For simplicity, we assume all the variable names are different and a program variable is associated with a namesake index variable. A type state refines a traditional product type with an index proposition. Intuitively, $(R; P)$ is a type for a machine state where the variables and registers satisfy the the proposition P. For example, a type for states where r_1 is greater than zero is $(R; x_1 > 0)$.

The other form of index-type annotations is called an indexed state transformer type. This concept is to treat a sequence of instructions as a function from one machine state to another. Intuitively, a state transformer type describes the computational effects of an instruction sequence. For example, the instruction `inc` r_0 has the effect that if $r_0 \geq 0$ before the instruction is executed, then $r_0 > 0$ afterward. In SDTAL's notation, this type is: $(R; \mathit{true}) \rightarrow (R_r; x_0 >= 0 \Rightarrow x_{0r} > 0)$. Generally, a state transformer type is of the form $(R; P_1) \rightarrow (R_r; P_2)$. R_r is of the form $(\mathtt{int}(x_{0r}), ..., \mathtt{int}(x_{(n-1)r}), \mathtt{var}_{1r} ...)$. It associates index variables $\{x_{0r}, ..., x_{(n-1)r}, \mathtt{var}_{1r}, ...\}$ with a machine state. These index variables refer to values in the state after the an instruction is executed(the post state). Proposition P_2 ($x_0 >= 0 \Rightarrow x_{0r} > 0$ in the example above), called a post condition, is over index variables $\{x_0, x_{0r}, ..., x_n, x_{nr}, \mathtt{var}_1, \mathtt{var}_{1r}, ...\}$. A post condition describes the relation between index variables from the state prior to the statement (pre state) and the post state.

A type state is associated with a label. Intuitively, a type state represents an invariant at that program location. A state transformer type is given to an instruction sequence. Later, a type state will be used to encode the `assume`-statement in a Boolean program; and a state transformer type will be used for a simultaneous assignment in a Boolean program.

Type checking an SDTAL program will generate a set of constraints, which are solved by a decision procedure. A type checker will infer a type for a location or a sequence of instructions; this involves the gathering of semantic information in data flow analysis and testing for subtyping relation. Note that there is no primary type in an index type system. Constraints are generated during subtyping test. Details follow in Section 4.

3 Certifying Compiler

In ACC, certification contains three steps: 1) compute an abstraction, 2) compile the source program and encode the abstraction, and 3) optimize the SDTAL program. The first step is covered by other authors [CGP99, Kur94, BR01, GS97]. We only need to assume the source language is similar to C or Java, so that an effective abstraction algorithm can be reused. In this section, we explain the second step. The third step is discussed at length in another paper [XH03b].

3.1 Encoding Boolean Programs in DTAL

Given a source program C, a corresponding Boolean program B and a type-less SDTAL program D, assume function E maps a B expression to its corresponding C expression. A mapping $\Theta(x)$ returns the index variable indexing the SDTAL variable corresponding to C variable x. We extend Θ over expressions in the natural way. An index proposition P_r means the proposition is evaluated in a post state, which means the index variables in P are replaced by those sub-scripted with "r". For example, proposition $(x_1 > x_2)_r$ is the same as $(x_{1r} > x_{2r})$.

For a C assignment S_C, suppose the corresponding Boolean assignments are of the form $b = H(e_1, e_2)$, and S_D is the assembly code compiled from S_C. According to the definition of an H function, we can encode the Boolean assignments as a state transformer state, which types S_D.

$$S_D : (R; \mathit{true}) \rightarrow (R; \bigwedge\{\Theta(E(e_1)) \Rightarrow \Theta(E(b))_r, \Theta(E(e_2)) \Rightarrow \Theta(E(\neg b))_r\})$$

where $\bigwedge$ is the set conjunction operator and ":" is the typing relation.

For an if-statement, an invariant (type state) encoding the assume condition that leads the then branch is specified at the then- dummy block. Another one, encoding the assume condition that leads the else branch is specified at the else-dummy block.

In the worst case, the size of the certificate is $O(m * 2^n)$, where n is the number of predicates and m is the size of the program. In comparison, a proof based certificate is proportional to the size of a search tree, which is exponential in the size of the program. In practice, n is a very small number; the number of annotations is far less than m. This is especially true for industrial strength programs. In Section.5, we will describe our experience.

4 Abstraction Validation

In this section, first we introduce the dynamic semantics of SDTAL. Then we present the type checking rules. Finally we state the soundness of the type system and relate this soundness to the correctness of the ACC system.

$$\frac{J(ic) = add\, r_a\, r_b\, r_i}{(ic, M) \hookrightarrow (ic + 1, M[i \mapsto (M(a) + M(b))])} \text{ (eval-add)}$$

$$\frac{J(ic) = \texttt{store}\, r_i\, \texttt{var}}{(ic, M) \hookrightarrow (ic + 1, M[\texttt{var} \mapsto M(i)])} \text{ (eval-store)}$$

$$\frac{J(ic) = \texttt{jmp}\, l}{(ic, M) \hookrightarrow (l, M)} \text{ (eval-jump)}$$

$$\frac{J(ic) = \texttt{jeq}\, l_1\, l_2\, r_i\, r_j \quad M(i) = M(j)}{(ic, M) \hookrightarrow (l_1, M)} \text{ (eval-jeq-yes)}$$

$$\frac{J(ic) = \texttt{jeq}\, l_1\, l_2\, r_i\, r_j \quad M(i)! = M(j)}{(ic, M) \hookrightarrow (l_2, M)} \text{ (eval-jeq-no)}$$

Figure 6: Selected Evaluation Rules for SDTAL

4.1 Dynamic Semantics

We will define a transition semantics for SDTAL, namely, a transition relation $\hookrightarrow^b$, which is later used in the account of the soundness. This relation is based on a set of evaluation rules for SDTAL, which runs on an abstract machine described below.

The abstract machine assumes a number (n) of registers and unlimited memory. The machine state is a pair $(\texttt{ic}, M)$, where $\texttt{ic}$ is the instruction counter and M maps registers and variables to integers. For example, $M(i)$ represents the i-th register and $M(x)$ represents the value of x. Let u range over registers and variables. We write $M[u \mapsto v]$ for a mapping where $M(u)$ is updated to v. We assume that $\texttt{ic}$ are integers and that $J(l)$ returns the instruction at location l.

The dynamic semantics is presented as a set of evaluation rules in Fig. 6. A typical rule is the (eval-add) rule, the antecedent of which says the current instruction is an addition. The consequent is a judgment $(ic, M) \hookrightarrow (ic', M')$ stating that the machine moves one step. This rule says the ic will increment by 1 and the target register is updated to hold the sum. The semantics of a conditional branch is given by, for example, rules `(eval-jeq-yes)` and `(eval-jeq-no)`.

We annotate $\hookrightarrow$ with the instruction that enables this state transition. These evaluation rules above also define a branch relation $\hookrightarrow^b$ for a block b. $(ic, M) \hookrightarrow^b (ic', M')$ if b is composed of instructions $I_1, I_2, ..., I_n$, such that $(ic, M) \hookrightarrow^{I_1} (ic_1, M_1) \hookrightarrow^{I_2} ... \hookrightarrow^{I_n} (ic_n, M_n)$.

Next we introduce the type checking procedure.

4.2 Type Checking

A typed SDTAL program D is composed of a set of blocks $\{b_i\}$. Each b_i has a label l_i with an invariant τ_{1i}, a body S_i of type τ_{2i}, which is of the form $R; P_i \to R_r; Q_i$, and a branch instruction j_i. We assume the labels are symbolic names of the addresses of the instructions. The type-checking is to establish that for every i: 1) S_i is of type τ_{2i}, and 2) when j_i jumps to a label l_x, propositions τ_{1x} and P_i always hold, for any possible x.

To check the first condition, we process one block body at a time. We will first infer a type of the body and then compare the inferred type with the annotations. Specifically, given a typing $(S : (R; P \to R_r; Q)))$, we will take P as the context and then use a set of rules to compute the post type state of this sequence. The final post type state is tested against the specified post state type for subtyping. As is typical with index-types, the subtyping test generates verification conditions, which are solved by a decision procedure. Below we define relation $\sqsubseteq$ (subtype of) over product index-types. Intuitively, an indexed product type is a subtype of another, if the proposition that constrains the candidate sub-type implies the proposition that constrains the candidate super type.

$$\frac{\models_d (P \Rightarrow P')}{(T; P) \sqsubseteq (T; P')} \text{ (tc-subtype)}$$

The judgment form $\vdash S : \tau$, where τ is a state transformer type, establishes the typing of a block body S. The judgment form $\phi, \theta \vdash S \Uparrow P' \Downarrow \theta'$ asserts the statements S, generating a current condition P' and a mapping θ' between index variables, under an index context ϕ and an initial θ. Intuitively, the second form of judgment computes some

$$\frac{(P_1) \vdash S \Uparrow (P_3) \Downarrow \theta \quad \vdash ((R, R_r); \iota(P_3, \theta)) \sqsubseteq ((R, R_r); P_2)}{\vdash S : ((R; P_1) \rightarrow (R_r; P_2))} \text{ (tc-block)}$$

$$\frac{\phi' = \phi[v_n/X_3], X_3 = (X_1 + X_2)[v_n/X_3] \quad \theta' = \theta[X_3 \mapsto v_n] \quad \phi', \theta' \vdash I \Uparrow \phi'' \Downarrow \theta''}{\phi, \theta \vdash (\texttt{add}\, R_1\ R_2\ R_3; I) \Uparrow \phi'' \Downarrow \theta''} \text{ (tc-add)}$$

$$\frac{\phi' = \phi[v_n/\texttt{var}], (\texttt{var} = X_1) \quad \theta' = \theta[X_1 \mapsto v_n] \quad \phi', \theta' \vdash I \Uparrow \phi'' \Downarrow \theta''}{\phi, \theta \vdash (\texttt{store}\, R_1\ \texttt{var}; I) \Uparrow \phi'' \Downarrow \theta''} \text{ (tc-store)}$$

$$\frac{(R; (\phi, X_1 = X_2)) \sqsubseteq (\tau_{1t}; P_t) \quad (R; (\phi, X_1 != X_2)) \sqsubseteq (\tau_{1e}; P_e)}{\phi \vdash \texttt{jeq}\, R_1\ R_2\ l_t\ l_e} \text{ (tc-jeq)}$$

Figure 7: Selected Typing Rules

intermediate result in typing a block, while the first form establishes the type of a block. A current condition is an index proposition that precisely describes the machine state after S. Intuitively, θ remembers the information about index variables corresponding to the values that have been modified since the beginning of S. Meta function ι, given a current condition P and the history θ, returns the post condition (with respect to the state before S, where θ is empty). Details and an example will be given below.

We often blur the distinction between ϕ and P. T is an unconstrained product type. We say $\models_d P$ if P is valid in a selected logic d.

In Figure 7 is a selected set of checking rules. For a sequence beginning with an add instruction, the (tc-add) rule computes a current condition ϕ' and a new mapping θ'. Both θ' and ϕ' are adopted as the context to check the rest of the sequence. When a variable (or register) is modified, we use a new index variable v_n to substitute the modified index variables in the context, as in $\phi[v_n/X_3]$. The reason is that we have to keep the information that involves the modified variable to correctly compute a strong current condition. For example, if x is modified and we know $y < x$ and $x < z$, then both of these propositions no longer hold. By renaming x to a new variable, we manage to keep the $<$ relation between y and z.

The initial value of X_3 is kept by v_n. If we want to compute the post condition from a current condition, the X_i's in the current condition are in fact X_{ir}'s. Remembering which index variable stores the initial value of X_i helps us to recover the initial value of X_i. In computing the history mapping θ, expression $\theta[X_3 \mapsto v_n]$ only updates X_3's mapping when X_3's mapping is not defined. Index variable v_n in the same rule refers to the same fresh new index variable. Therefore, we are able to define the ι function as:$\iota(P, \theta) = P_r[\theta^{-1}] \wedge \texttt{id}_{\hat{\theta}}$ where $\texttt{id}_{\hat{\theta}}$ returns equalities among the index variables corresponding to values that are not changed. In the context of predicate abstraction, $\texttt{id}_{\hat{\theta}}$ can be represented by only a few equations.

As an example, for a statement $\texttt{add}\ r_0\ 1\ r_0$, we will have $x_0 = v_1 + 1$ in the computed post condition and a θ mapping x_0 to v_1. When this condition is used as the post condition of a block, to compare with the specified post type state, we compute $P_{3r}[\theta^{-1}]$ (as in the (tc-block) rule). The component $x_0 = v_1 + 1$ now becomes $x_{0r} = x_0 + 1$.

In SDTAL, branches are always at the end of a block. If the block is an assignment block, then we test if the initial invariant and the precondition of the target (usually the next) block is a super type of the state type before the jump. If the block is a conditional test, we add the then-condition to the context to check the initial invariant and the precondition then- dummy block; and we add the negation of that jump condition to the context to verify the invariant and the precondition of the else- dummy block.

Given a program D as defined in the beginning of the section, if $\forall i$, there exists ϕ_i such that $\vdash (S_i : \tau_{2i}) \Uparrow \phi_i$ and $\phi_i \vdash j_i$, then we say the program P is well typed.

4.3 Soundness

Below we state the soundness of the typing rules of SDTAL with respect to the $\hookrightarrow^b$ transition semantics. The soundness theorem states that if a program is well typed, and if the machine moves from one state to another by executing a block, then these two states satisfy the annotated post type state; and the computed post state also satisfies the invariant at the entry of the a target block.

Specifically, (M, M') is a product of two machine states. They as a whole can be viewed as a mapping from registers and variables to integer values. Remember Θ maps a program variable to the corresponding index variable. Therefore, $(M, M')[\Theta]$ is a mapping from index variables (from both states) to integer values. For example, if $M(r_0) = 0$ and

	Program Size	Certificate Size	Percentage
Bounded Buffer	702	232	33%
Vector Addition	808	308	38%
Reader/Writer	1570	282	18%
Sleeping Barber	809	200	25%

Figure 8: Size of Certificate in Bytes

$M'(r_0) = 1$, we will have $(M, M')[\Theta](x_0) = 0$ and $(M, M')[\Theta](x_{0r}) = 1$. We write $(M, M')[\Theta] \models P$ if P is evaluated to be true in the integer domain, given the mapping of $(M, M')[\Theta]$. For example, in the example above, we have: $(M, M')[\Theta] \models x_{0r} = x_0 + 1$.

Theorem 4.1 *Let D be a well typed program composed of blocks b_i's. Let the label of b_i be l_i and let the state transformer type of body of b_i be $(R; P_i \rightarrow R_r; Q_i)$. For any initial machine state M_0, if $(l_1, M_0) \hookrightarrow^* (l_x, M)$, then either the machine stops, or for all l_y such that $((l_x, M) \hookrightarrow^{b_i} (l_y, M'))$, it holds that $(M, M')[\Theta] \models (Q_x)$ and $M'[\Theta] \models \tau_{1y} \wedge P_y)$.*

This theorem implies that the Boolean program encoded by the type annotations is valid if the code is type correct. In other words, it establishes the correspondence between an assembly program and a Boolean program. In a sister paper, we use control flow automata (CFA) to formulate the source, the Boolean and the SDTAL program. That paper also argues for the preservation of this CFA during a compilation process identical to the one described in this paper. A CFA can induce a transition system on which the semantics of LTL formulas are defined. Induction on the form of LTL formula shows that if an abstract program model checks, so will any program sharing the same CFA. Here we restate the result in the context of this paper.

Proposition 4.2 *Let $D = (l_1\ (b_1 : \tau_1), l_2\ (b_2 : \tau_2), ..., l_n\ (b_n : \tau_n))$ be a program. Assume $\forall i < n, \vdash b_i : \tau_i$. If $B = \texttt{decode}(\tau_1, ..., \tau_n)$, and $B \models L$, then $P \models L$, for any next operator free LTL formula L.*

The function `decode` calculates the Boolean program from the index-type annotations.

5 Experience

We have implemented Boolean abstraction for a higher-level while language with concurrent extensions. We have applied this tool to several well-studied concurrent programs found in a textbook[And91], including:

- Readers/Writers program: Many but fixed number of readers and writers randomly generate requests to read/write a database. They are synchronized in such a way that at any time there is at most one writer in the critical section.
- Sleeping Barber program: A fixed number of customers repeatedly enter a barber's shop. They synchronize with each other and with the only barber to be served. We verify that any customer who enters the room will be served.

In Table 8 we list the size of the index-type annotations (certificate), relative to the size of the SDTAL program (code plus annotation). The numbers are based on the sizes of the assembly programs generated by a prototype compiler for the while language.

Recently, we applied ACCEPT to the reachability test of program locations for C programs. We leverage BLAST [HJM+02], calculating a Boolean program from BLAST's intermediate result. This way, we take advantage of the lazy abstraction technique used by BLAST. The encoder of Boolean program and the SDTAL type checker are reused. We are able to generate small size certificate for industrial strength programs. Details of this experience are described in another paper [XH03b].

6 Conclusion

The contribution of this paper is: 1) we propose an implementation method for abstraction-carrying Code, and 2) we demonstrate that an index-typed assembly language can encode a Boolean program and help a client to validate it easily. Our prototype toolkit suggests potential applications of our approach and demonstrates its effectiveness.

This paper implicitly assumes that there is only one temporal property of concern. However, one of the most appealing aspects of the ACC framework is that one abstraction may allow a client to check a cluster of temporal properties. For example, the client may choose to check the mobile code for different aspects regarding file operations. The server may not know which one in this cluster of the temporal properties the client checks. The server calculates a "sum" abstraction of several abstractions. The computation of each of the small abstraction is still guided by an individual temporal property. The client may only validate and model check part of the abstraction that is relevant to the properties it concerns. We intend to test this idea on some realistic programs.

Bibliography

[And91] Gregory Andrews. *Concurrent Programming*. Addison and Wesley, 1991.

[App01] Andrew Appel. Fundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science*, June 2001.

[BL02] A. Bernard and P. Lee. Temporal Logic for Proof-carrying Code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Danmark, July 2002.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV'98, LNCS 1427*, pages 319–331, June 1998.

[BR01] Thomas Ball and Sriram Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science,LNCS2057*, pages 103–122. Springer-Verlag, May 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Shawn Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Extract Finite-state Models from Java Source Code. In *Proceedings of ICSE 2000*, 2000.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[DDP99] Satyaki Das, David Dill, and Seung Joon Park. Experience with Predicate Abstraction. In *Proceedings of CAV'99, LNCS 1633*, pages 160–171, Trento, Italy, July 1999.

[GS97] Susanne Graf and Hassen Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Lecture Notes in Computer Science, LNCS 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[HJM+02] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. "Temporal-Safety Proofs for Systems Code. In *Computer-Aided Verification*, pages 526–538, 2002.

[HP00] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[Koz98] Dexter Kozen. Efficient Code Certification, 1998. Technical Report, Computer Science Department, Cornell University.

[Kur94] Robert Kurshan. Models Whose Checks Don't Explode. In *Computer-Aided Verification*, pages 222–233, 1994.

[MCGW98] Greg Morrisett, Karl Crary, Neil Glew, and David Walker. Stacked-based Typed Assembly Language. In *Proceedings of workshop on Types in Compilation, also in Lecture Notes in Computer Science, LNCS 1473*, pages 28–52. Springer Verlag, 1998.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Nam01] Kedar S. Namjoshi. Certifying Model Checkers. In *13th Conference on Computer Aided Verification (CAV)*, 2001.

[Nam03] Kedar S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *VMCAI*, 2003.

[Nec98] George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[Sai00] Hassen Saidi. Model-checking Guided Abstraction and Analysis. In *Proceedings of SAS'00, Lecture Notes in Computer Science, LNCS 1824*, pages 377–389, Santa Barbara, CA, USA, July 2000. Springer-Verlag.

[SRRS01] R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model-carrying Code(MCC): A New Paradigm for Mobile Code Security. In *New Security Paradigm Workshop*, 2001.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. Type System for Certified Binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, January 2002.

[TC01] Li Tan and Rance Cleaveland. Evidence-based model checking. In *13th Conference on Computer Aided Verfication (CAV)*, pages 455–470, 2001.

[XH01] Hongwei Xi and Robert Harper. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, September 2001.

[XH03a] Songtao Xia and James Hook. Abstraction-based Certification of Temporal Properties. In *Proceedings of Workshop on Formal Techniques for Java-like Programs*, 2003. To Appear.

[XH03b] Songtao Xia and James Hook. Experience with abstraction-carrying code, 2003. Submitted to Software Model Checking Workshop.

[Xi97] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.

[Zen98] Christopher Zenger. *Indizierte Typen*. PhD thesis, University of Karlsruhe, 1998.

A Synopsis of Soundness Proof

In this section, we prove the soundness theorem in Section 4.3.

We restate the theorem below.

Theorem A.1 *Let P be a well typed program composed of blocks b_i's. Let the label of b_i is l_i and the state transformer type of body of b_i is τ_i. Then for any initial machine state M0, if $(l_1, M0) \hookrightarrow^* (l_x, M)$, then either the machine stops, or $\forall l_y$ so that $((l_x, M) \hookrightarrow (l_y, M'))$, $(M, M')[\theta_i] \models \texttt{post}(\tau_x)$ and $M'[\theta_i] \models \delta(l_y)$.*

To prove this theorem, we first prove the lemma below (Lemma 1). Intuitively, this lemma states the one-step soundness of the type checking rules. For a non-branch instruction i, the soundness means the post condition its type checking rule calculates is valid in any machine state related by $\hookrightarrow^i$. For a branch instruction, the soundness indicates that the invariants stated wherever the next instruction locates, is a valid invariant for any possible machine state at that point.

We define a function δ that maps program locations (not necessarily labels) into index propositions for a well-typed program. If a program is well typed, then for all blocks b, there exists a derivation $\vdash b$. An important result of SDTAL is that such derivation is unique. Therefore, for every non-branch instruction preceding location l, there is an unique ϕ, P, θ and θ' so that $\phi, \theta \vdash J(l) \Uparrow P \Downarrow \theta'$. We define $\delta(l)$ as $P_r[\theta^{-1}]$. If a location is not preceded by a non-branch instruction, it must be the beginning of a target block, there must exists a branch instruction j, a ϕ and a θ so that $\phi, \theta \vdash j l_1 l_2$, where one of l_1 and l_2 is l.

Lemma A.2 *Let P be a well typed program composed of blocks $\mathtt{b}_i$'s. Let the label of $\mathtt{b}_i$ is l_i and the state transformer type of body of $\mathtt{b}_i$ is τ_i. Then for any initial machine state M_0, if $(l_1, M_0) \hookrightarrow^{I*} (pc, M)$, then either* $\mathtt{pc} = 0$ *(indicating halt), or for all pc' so that* $((\mathtt{pc}, M) \hookrightarrow^{J(pc)} (\mathtt{pc}', M'))$, $(M, M')[\theta_i] \models \delta(\mathtt{pc})$.

The proof of Lemma 1 is by induction on the number of moves. The induction step is proved by induction on the form of the instruction $J(\mathtt{pc})$. A few representative cases are listed below.

- $J(\mathtt{pc})$ is an addition add $R_1\ R_2\ R_3$: We write $\delta(\mathtt{pc})$ as $\phi, X_{3r} = X_2 + X_1, x_i = x_{ir}$ for all x_i except X_3. According to dynamic semantics, both $X_{3r} = X_2 + X_1$ and $x_i = x_{ir}$ are easily proved. And if the program is well typed, either ϕ is a precondition or ϕ is the post condition of a previous typing rule, both of which are proven by the I.H.
- $J(\mathtt{pc})$ is jeq $R_1\ R_2\ l_1\ l_2$: We write $\delta(\mathtt{pc}$ as an implication of either $\phi, X_1 = X_2$ or $\phi, X_1! = X_2$. Take $\phi, X_1 = X_2$. $X_1 = X_2$ is proven by the dynamic semantics and ϕ is from the I.H.

Symbolic Approach to the Analysis of Security Protocols*

Stéphane Lafrance John Mullins

Department of Computer and Software Engineering
École Polytechnique de Montréal
Montréal, Canada
{Stephane.Lafrance,John.Mullins}@polymtl.ca

Abstract

The specification and validation of security protocols often requires viewing function calls – like encryption/decryption and the generation of fake messages – explicitly as actions within the process semantics. Following this approach, this paper introduces a symbolic framework based on value-passing processes able to handle symbolic values like fresh nonces, fresh keys, fake address and fake messages. The main idea in our approach is to assign to each value-passing process a formula describing the symbolic values conveyed by its semantics. In such symbolic processes, called *constrained processes*, the formulas are drawn from a logic based on a message algebra equipped with encryption, signature and hashing primitives. The symbolic operational semantics of a constrained process is then established through semantic rules updating formulas by adding restrictions over the symbolic values, as required for the process to evolve. We then prove that the logic required from the semantic rules is decidable. We also define a bisimulation equivalence between constrained processes; this amounts to a generalisation of the standard bisimulation equivalence between (non-symbolic) value-passing processes. Finally, we provide a complete symbolic bisimulation method for constructing the bisimulation between constrained processes.

1 Introduction

The sudden expansion of electronic commerce has introduced an urgent need to establish strong security policies for the design of security protocols. The formal validation of security protocols has since became one of the primary tasks in computer science. In recent years, equivalence-checking has proved to be useful for the verification of security protocols [AG98, BNP99, Cor02, LM02]. The main idea behind this approach of formal verification is to verify a security property by testing whether the process (specifying the protocol) is bisimilar to its intended behaviour. The success of these methods relies on two facts: 1) process algebras are suitable for the specification of such protocols, including cryptographic protocols; 2) bisimulation offers an expressive semantics to process calculi. Many other methods inspired by a wide range of approaches have been proposed in the literature to analyse security protocols, but very few offer the possibility to explicitly analyse function calls used for example in encrypting, decrypting, signing and hashing. In cryptographic based process calculi like Abadi & Gordon's *spi-calculus* [AG99] and Focardi & Martinelli's *CryptoSPA* [FM99], encryption and decrypting manipulations are done in a parallel inference system, and therefore they are not directly observable from the process semantics. For instance, a principal sending a message m encrypted with a key k is modeled as an action "$\overline{c}(\{m\}_k)$" (where $\{m\}_k$ stands for the message m encrypted by k) whenever $\{m\}_k$ can be inferred from the principal's current knowledge.

But information flow properties (e.g. non-interference [FG95] and admissible interference [Mul00]) usually require such manipulations to be observable. For that purpose, we work within the frameork of an extension of value-passing CCS [Mil89], called *Security Protocols Process Algebra* (SPPA) [LM03], in which function calls made by principals are explicitly modeled as actions. For instance, a principal sending a message m encrypted with a key k is modeled as the action "$\{m\}_k := enc(k, m)$" (the principal encrypting the message) followed by the output action "$\overline{c}(\{m\}_k)$". Moreover, the specification of intruders in SPPA allows us to analyse the effects on the information flow of a protocol of an intruder generating fake addresses and fake messages. In addition, compared with a process calculus using an inference system for encryption manipulations, SPPA is more suited for analysing restricted attacks based on the

*The first author has been supported by an FCAR doctoral scholarship (Quebec Government) and the second, by the NSERC grant no. 138321-01 (Canadian Government).

repetition of the same attempt. For instance, distributed denial of service attacks have been specified in SPPA [LM03]. In order to deal with the notion of fake message, around which most attacks are built, we need to extend SPPA in order to specify functions generating random values. But the introduction of such generating function calls as actions requires interpreting their output as symbolic values. Thus, we need to consider symbolic value-passing processes along with a symbolic operational semantics able to handle symbolic variables without a specific value but satisfying certain constraints.

This paper introduces a symbolic framework for the specification of security protocols which is based on the novel concept of *constrained process*. A constrained process is a pair composed of a value-passing process (SPPA process with, possibly, free variables standing for symbolic values) and a formula expressing a statement about symbolic values. The formula pertains to a message logic whose terms are taken from a message algebra relying on atomic sets of numbers and identifiers (addresses), and cryptographic primitives (encryption, signing and hashing operators). Therefore, the purpose of a formula within a constrained process is to bind the free variables occurring in the course of process execution. For instance, to a process generating a fresh key for a protocol run and allocating this key to some free variable x, we assign the formula which states that x stands for a key. The operational semantics of constrained processes is thus achieved from the process behaviour, subject to the restrictions imposed by its formula. Hence, a process whose definition requires the execution of an action and evolution into another process will only occur if the whole transition satisfies the formula enforced at this point. Roughly speaking, the formula within a constrained process stands for the set of messages that can be assigned to its free variables; this set of possible values evolves, along with the process, by either adding new free variables or restricting (or binding) the ones already present. In one of the main result of this paper, we prove the decidability of every formula derivablefrom the process algebra's operational semantics. Moreover, we feel that our symbolic framework can be applied to any other process algebra, from value-passing CCS to more expressive process algebras like Milner's π-*calculus* [MPD92] and *spi-calculus*.

The use of value-passing processes over infinite messages-domain leads to non finite-branching transition systems on which bisimulation equivalence fails to be decidable. An attractive solution to this challenge was proposed by Hennessy-Lin [HL95] who defined a notion of symbolic bisimulation. It is primary based on a symbolic semantic which may express value-passing CCS processes in terms of finite symbolic transition systems instead of possibly infinite ones. The main idea behind Hennessy-Lin's approach is to assign to every action (transition) a formula describing the symbolic values (free variables) used in the action. Within this framework, they introduce two generalisations of Milner's strong bisimulation equivalence for value-passing processes called *early* and *late* bisimulation. Although our paper aims at a similar goal, we introduced a symbolic semantics in which the description of symbolic values is done within the processes instead of within the transitions. In fact, our symbolic transition graphs could be directly obtained from their symbolic transition graphs (by considering every path). Moreover, our approach, compared to Hennessy-Lin's, takes advantage of an expressive message logic capable of stating cryptographic relations. For instance, we can bind free variables x_1, x_2, x_3 through the formula $(x_1 == \{x_2\}_{x_3}) \wedge \mathcal{K}(x_3)$ which states that x_1 stands for x_2 encrypted with the key x_3. In addition, we feel that the concept of constrained process is more suited for security protocol analysis than Hennessy-Lin's symbolic transition graph: a constrained process allows us to get a quick view of the symbolic values at a given state of the protocol, rather than retrieving successively every path leading to this state.

In section 2, we introduce a logic for cryptographic messages. In section 3, we present the SPPA process algebra and in section 4 we describe a symbolic semantics for constrained processes. We conclude this paper with a short talk on related work and on our future work.

2 Message Specification

2.1 Message Algebra

We consider the following message algebra which relies on disjoint syntactic categories of principal identifiers, variables and numbers respectively ranging over sets $\mathcal{I}, \mathcal{V}$ and $\mathcal{N}$. The set of *terms* $\mathcal{T}$ is constructed as follows:

$$\begin{array}{lllllllllll} t & ::= & n & (\textit{number}) & \mid & id & (\textit{identifier}) & \mid & x & (\textit{variable}) & \mid \ (t,\ldots,t) \ \ (\textit{n-tuple}) \\ & \mid & \{t\}_t & (\textit{encryption}) & \mid & [t]_t & (\textit{signature}) & \mid & h(t) & (\textit{hashing}). & \end{array}$$

It is important to note that we only consider finite terms. For any term t, we denote $fv(t)$ the set of variables occurring in t. The set $\mathcal{M}$ of *message* contains all term t such that $fv(t) = \emptyset$. Furthermore, given a valuation $\varrho : \mathcal{V} \to \mathcal{M}$ and a term t such that $fv(t) = \{x_1, \ldots, x_n\}$, $\varrho(t)$ stands for the message $a[\varrho(x_1)/x_1] \ldots [\varrho(x_n)/x_n]$ i.e. the message obtained from t by substituting each variable x_i with its valuation $\varrho(x_i)$ $(i = 1, \ldots, n)$.

For the sake of clarity, we will discriminate a subset $\mathcal{K} \subseteq \mathcal{M}$ of messages that may be used as encryption keys. Note that the definition of the set $\mathcal{K}$ usually depends on the cryptosystems used by the protocol. For instance, in the case of a symmetric block-cypher algorithm, we have $\mathcal{K} = \{v \in \mathcal{N} \mid |v| = N\}$ for some N. For generality purposes, this paper uses the set $\mathcal{K} = \mathcal{N} \cup \bigcup_{m \geq 1}\{h^m(n) \mid n \in \mathcal{N}\}$, where we write $h^m(n)$ instead of $h(\ldots h(n)\ldots)$ (m times). Moreover, in order to deal with public-key encryption, we use an idempotent operator $[-]^{-1} : \mathcal{K} \to \mathcal{K}$ such that a^{-1} denotes the private decryption key corresponding to the public encryption key a, or vice versa. For symmetric encryption, let $a^{-1} = a$. It is assumed that there are perfect encryption and hashing.

2.2 A Logic for Messages

In the following, we consider the logic based on the terms of our message algebra and the following *predicates*:

$$\begin{array}{rcllcll} \mathcal{P} & ::= & t == t & (\textit{term equation}) & \mid & \mathcal{M}(t) & (\textit{message predicate}) \\ & \mid & \mathcal{N}(t) & (\textit{number predicate}) & \mid & \mathcal{K}(t) & (\textit{key predicate}). \end{array}$$

The *formulas* of our logic are then obtained as follows:

$$\phi \quad ::= \quad \mathbf{0} \quad (\textit{false}) \quad \mid \quad \mathbf{1} \quad (\textit{true}) \quad \mid \quad \mathcal{P} \quad (\textit{predicate}) \quad \mid \quad \phi \wedge \phi \quad (\textit{conjunction}) \quad \mid \quad \exists_x\, \phi \quad .$$

The set of ϕ's free variables is denoted by $fv(\phi)$ and ϕ is said to be closed whenever $fv(\phi) = \emptyset$. The satisfaction of a closed formula ϕ, denoted by $\models \phi$, is defined recursively as follows:

- $\models\ a == b$ iff messages a and b are syntactically identical i.e.,
  - $\models\ n == n$ for every $n \in \mathcal{N}$,
  - $\models\ id == id$ for every $id \in \mathcal{I}$,
  - $\models\ (a_1, a_2) == (b_1, b_2)$ iff $\models\ a_1 == b_1 \ \wedge\ a_2 == b_2$,
  - $\models\ \{a_2\}_{a_1} == \{b_2\}_{b_1}$ iff $\models\ a_1 == b_1 \ \wedge\ a_2 == b_2$,
  - $\models\ [a_2]_{a_1} == [b_2]_{b_1}$ iff $\models\ a_1 == b_1 \ \wedge\ a_2 == b_2$, and
  - $\models\ h(a) == h(b)$ iff $\models\ a == b$;
- $\models\ \mathcal{M}(a)$ for every message $a \in \mathcal{M}$;
- $\models\ \mathcal{N}(a)$ iff $a \in \mathcal{N}$;
- $\models\ \mathcal{I}(a)$ iff $a \in \mathcal{I}$;
- $\models\ \mathcal{K}(a)$ iff $a \in \mathcal{K}$.

We assume that each predicate is decidable i.e., the satisfiability problems $\models \mathcal{N}(a)$, $\models\ \mathcal{I}(a)$ and $\models \mathcal{K}(a)$ are decidable for any $a \in \mathcal{M}$. Note that the predicates $\mathcal{I}(a)$ and $\mathcal{N}(a)$ are never satisfied when a is a non-atomic message. For instance, $\not\models \mathcal{I}(h(a))$ and $\not\models \mathcal{N}(\{a\}_b)$ for any $a, b \in \mathcal{M}$. As for predicate $\mathcal{K}$, $\models \mathcal{K}(a)$ if and only if $a = h^m(n)$ for some $n \in \mathcal{N}$ and $m \geq 0$. Moreover, we always assume that $\mathcal{I}$ and $\mathcal{N}$ are disjoint sets. The satisfaction of non-atomic formulas $\phi \wedge \phi$ and $\exists_x\, \phi$ are defined in the natural way.

Given a valuation $\varrho : \mathcal{V} \to \mathcal{M}$, the satisfaction of a formula ϕ by ϱ, denoted by $\varrho \models \phi$, is defined as follows:

$$\varrho \models \phi \quad \text{iff} \quad \models \varrho(\phi).$$

Two formulas ϕ and ϕ' are said to be *equivalent* – which denoted by $\phi \Leftrightarrow \phi'$ – whenever

$$\varrho \models \phi \Longleftrightarrow \varrho \models \phi'$$

for every valuation ϱ. In particular, a formula ϕ is said to be equivalent to $\mathbf{0}$ – which is denoted by $\phi \Leftrightarrow \mathbf{0}$ – whenever $\varrho \not\models \phi$ for every valuation ϱ.

2.3 Decidability

Our goal is to prove that every (closed) formula from our logic is decidable. This result follows from the fact that our logic for messages is restricted to conjunction and existential operators. Hence, the decidability of a formula boils down to the satisfaction of a number of predicates and equations, which we assume to be decidable.

Theorem 2.1 *Every formula is decidable.*

Proof of Theorem 2.1 is given in Appendix B.

2.4 Private Functions

We consider a finite set $\mathcal{F}$ of *private functions* ranging over messages and allowing us to create new messages using the grammar rules above. Each private function f has a *characterisation formula* from our logic, denoted by ϕ_f, which is satisfied only by messages within its domain. Therefore, $\models \phi_f(a)$ if and only if $f(a)$ is defined (where we write $\phi_f(a)$ instead of $\phi_f[a/x]$ with $fv(\phi_f) = \{x\}$). In addition, we consider the notion of *generating function*, extremely useful for the specification of security protocols requiring fresh nonces, fresh keys and random numbers, or the specification of intruders generating fake addresses and fake messages. Generating functions are private functions which may generate symbolic values without any input. Each generating function $new(-)$ is assigned to a formula ϕ_{new}, also called characterisation formula, which is satisfied only by messages within its range. Moreover, we assume disjoint sets $\mathcal{F}_{id}$ of private functions, for every identifier id, such that $\mathcal{F} = \bigcup_{id\in\mathcal{I}} \mathcal{F}_{id}$. Intuitively, the principal assigned to the identifier id has only access to functions from $\mathcal{F}_{id}$, which usually contains the following:

- $extract^{n,i}_{id}((a_1,\ldots,a_n)) = a_i$ with $\phi_{extract^{n,i}_{id}}(x) ::= \exists_{x_1,\ldots,x_n} x == (x_1,\ldots,x_n)$;
- $enc_{id}(k,a) = \{a\}_k$ with $\phi_{enc_{id}}(x) ::= \exists_{x_1,x_2}(x == (x_1,x_2) \wedge \mathcal{K}(x_1))$;
- $dec_{id}(k^{-1},\{a\}_k) = a$ with
  $\phi_{dec_{id}}(x) ::= \exists_{x_1,x_2,y}(x == (x_1,x_2) \wedge \mathcal{K}(x_1) \wedge x_2 == \{y\}_{x_1})$;
- $hash_{id}(a) = h(a)$ with $\phi_{hash_{id}}(x) ::= \mathcal{M}(x)$;
- $sign_{id}(k,a) = [a]_k$ with $\phi_{sign_{id}}(x) ::= \exists_{x_1,x_2}(x == (x_1,x_2) \wedge \mathcal{K}(x_1))$;
- $newMessage_{id}(-)$ with $\phi_{newMessage}(x) ::= \mathcal{M}(x)$;
- $newNumber_{id}(-)$ with $\phi_{newNumber}(x) ::= \mathcal{N}(x)$;
- $newId_{id}(-)$ with $\mathrm{im}(newId)(x) ::= \mathcal{I}(x)$;
- $newKey_{id}(-)$ with $\phi_{newKey}(x) ::= \mathcal{K}(x)$.

Note that function $checksign$ does not produce new terms since its primary task is to verify whether its entry term is within its domain. Such a private function, called *verification function*, is treated as a function with a unique output – the Boolean "$(k^{-1},a,[a]_k) \in \mathrm{dom}(checksign_X)$".

3 Security Protocol Process Algebra

The first step in the validation of security protocols is to find a language able to express both the protocols and the security properties we want to enforce. Process algebras have been used for some years to specify protocols as a cluster of concurrent processes able to communicate in order to exchange data. Process algebras CSP [Low96, Sch96] and value-passing CCS [Mil89] have been extensively used with this objective. In this section, we introduce a symbolic framework which we feel could be applied to numerous process algebra. Given a process algebra, we proceed by extending its syntax in order to view generating functions call $x := new(-)$ as prefixes. Messages generated this way can be explicitly typed with a characterisation formula. For simplicity, our symbolic framework follows *Security Protocols Process Algebra* (SPPA) [LM03], an extension of value-passing CCS in which local functions calls are viewed as visible actions.

3.1 Syntax of SPPA

We consider a finite set C of *public channels*. Commonly public channels are used to specify message exchanges between principals. We assume that public channels have no specifics domains: any message can be sent or received over them. SPPA *prefixes* are obtained as follows:

$$\begin{array}{rrlll} \mu & ::= & \overline{c}(t) \quad (\textit{output prefix}) & \mid & c(x) \quad (\textit{input prefix}) \\ & \mid & x := new(-) \quad (\textit{generating prefix}) & \mid & x := f(t) \quad (\textit{functional prefix}) \end{array}$$

where t is any term such that $x \notin fv(t)$. From this syntax, we see that symbolic values generated by generating functions are specified as variables.

Let μ be a prefix and let t, t' be terms. The *agents* of SPPA are constructed from the following grammar:

$$\begin{array}{rrllllll} S & ::= & \mathbf{0} & (\textit{empty agent}) & \mid\ \mu.S & (\textit{prefix agent}) & \mid\ [t = t']\,S & (\textit{match}) \\ & \mid & S\backslash L & (\textit{restriction}) & \mid\ S|S & (\textit{parallel composition}) & \mid\ S + S & (\textit{sum}) \\ & \mid & S/\mathcal{O} & (\mathcal{O}\textit{-observation}) & & & & \end{array}$$

where L is a set and $\mathcal{O}$ is a partial mapping (both clarified below in Sect. 3.2). From this syntax, recursion is handled via agent names (e.g. by writing $S = \mu_1.\mu_2.S$). Also, in order to prevent name clashes for variables, we assume that a variable is never used twice to define agents. Given an agent S, we define its set of *free variables*, denoted by $fv(S)$, as the set of variables x appearing in S which are not in the scope of an input prefix $c(x)$, a functional prefix $x := f(t)$ or a generating prefix x := new (-); otherwise the variable x is said to be *bound*. Given a free variable $x \in fv(S)$ and a term t, we consider the substitution operator $S[t/x]$ where every free occurrence of x in S is set to t. A *closed agent* is an agent S such that $fv(S) = \emptyset$.

Intuitively, closed agents are used to specify principals of security protocols. More specifically, a SPPA *principal* is a pair (S, id) where S is a agent and $id \in \mathcal{I}$ is an identifier. The purpose of this notation is to relate a SPPA agent S and its sub-agents, to their unique owner (principal) via its identifier id. Moreover, given a principal A from a protocol, we frequently use the identifier id_A for a message containing its address, while we simply use A to refer to the protocol's entity. For simplicity, we often use abbreviated notations: $A_1|A_2$ instead of $(S_1|S_2, id)$, $A_1 + A_2$ instead of $(S_1 + S_2, id)$, and $[t = t']A_1$ instead of $([t = t']S_1, id)$, where $A_1 = (S_1, id)$ and $A_2 = (S_2, id)$ (they must have the same identifier).

In order to specify a security protocol in SPPA, we use the classic approach [FGC97, Sch96] of specifying the principals as concurrent agents. Given the principals $A, A_1, \ldots, A_n$, *processes* are constructed as follows:

$$\begin{array}{rrllll} P & ::= & A & (\textit{principal}) & \mid\ A_1 \parallel \cdots \parallel A_n & (\textit{protocol}) \\ & \mid & P\backslash L & (\textit{restriction}) & \mid\ P/\mathcal{O} & (\mathcal{O}\textit{-observation}) \end{array}$$

where $\parallel$ is an associative and commutative operator that forces communication over public channels. We say that a process P is *closed* whenever $fv(P) = \emptyset$.

A *constrained process* is a pair $\langle P,\ \phi\rangle$ where P is a process and ϕ is a formula designed to constrain the free variables occurring in P. If ϕ and ϕ' are equivalent and have the same free variables, then the constrained processes $\langle P,\ \phi\rangle$ and $\langle P,\ \phi'\rangle$ are considered to be the same. Therefore, notation $\langle P,\ \phi\rangle$ really stands for $(P,\ \{\phi' \mid \phi' \Leftrightarrow \phi \text{ and } fv(\phi) = fv(\phi')\})$.

3.2 Symbolic Semantics

The value-passing operational semantics of a SPPA process is defined in Appendix A. Note that this value-passing semantics is only defined for closed processes. Also note that, because of the value-passing semantics, the obtained transition graph could be infinite. In this section, we establish a symbolic operational semantics for constrained processes, which correspond to finite labeled transition graphs.

Given a term t, the *actions* of SPPA are defined as follows:

$$\alpha \quad ::= \quad \mu \quad (\textit{prefix}) \quad \mid \quad \delta(t) \quad (\textit{marker action}) \quad \mid \quad \tau \quad (\textit{silent action}).$$

The silent action τ is used to express non-observable behaviours. Markers are introduced in an attempt to establish an annotation to the semantics of an SPPA process; they do not occur in the syntax of processes since they are not considered as prefixes, and their specific semantics restrict their occurrence in order to tag communications between principals. In value-passing process algebra, communication is commonly expressed by replacing the matching output action and input action by the silent action τ, causing a drastic loss of information in terms of the exchanged values and the parties involved. A marker action has three parameters: a principal identifier, a channel and a message. Roughly speaking, the occurrence of an *output marker* $\overline{\delta^c_{id}}(a)$ stands for "the principal id has sent message a over the channel c", and the occurrence of an *input marker* $\delta^c_{id}(a)$ stands for "the principal id has received message a over the channel c". Hence, similarly to private functions, every marker action belongs to the principal stated in its parameter. We often use C to denote both the set of public channels and the set of output and input actions.

Symbolic operational semantics for constrained processes is given in Fig. 1. It is inspired by Hennessy-Lin's symbolic operational semantics [HL95] where Boolean values guarding actions are replaced by formulas ϕ restricting free variables within the process.

`Input` $$\frac{-}{\langle c(x).P,\,\phi\rangle \xrightarrow{c(x)} \langle P,\,\exists_x\phi \wedge \mathcal{M}(x)\rangle}$$

`Output` $$\frac{-}{\langle \overline{c}(t).P,\,\phi\rangle \xrightarrow{\overline{c}(t)} \langle P,\,\phi\rangle}$$

`Function` $$\frac{f \in \mathcal{F}_{id_P}}{\langle x{:=}f(t).P,\,\phi\rangle \xrightarrow{x:=f(t)} \langle P,\,\exists_x\phi \wedge \phi_f(t) \wedge x{==}f(t)\rangle}$$

`Generator` $$\frac{new \in \mathcal{F}_{id_P}}{\langle x{:=}new(-).P,\,\phi\rangle \xrightarrow{x:=new(-)} \langle P,\,\exists_x\phi \wedge \phi_{new}(x)\rangle}$$

`Match` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\rangle \quad \text{and} \quad \psi \not\Leftrightarrow \mathbf{0}}{\langle [t{=}t']P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\psi\rangle}$$

`Sum` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\rangle \quad \text{and} \quad fv(\phi)\cap fv(\psi)=\emptyset}{\langle P{+}Q,\,\phi\wedge\psi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\wedge\psi\rangle}$$

`Parallel` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\rangle \quad \text{and} \quad fv(\phi)\cap fv(\psi)=\emptyset}{\langle P|Q,\,\phi\wedge\psi\rangle \xrightarrow{\alpha} \langle P'|Q,\,\phi'\wedge\psi\rangle}$$

`Protocol` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\rangle, \quad \alpha\notin C \quad \text{and} \quad fv(\phi)\cap fv(\psi)=\emptyset}{\langle P\|Q,\,\phi\wedge\psi\rangle \xrightarrow{\alpha} \langle P'\|Q,\,\phi'\wedge\psi\rangle}$$

`Synchronisation` $$\frac{\langle P,\,\phi\rangle \xrightarrow{c(x)} \langle P',\,\phi'\rangle, \quad \langle Q,\,\psi\rangle \xrightarrow{\overline{c}(t)} \langle Q',\,\psi'\rangle \quad \text{and} \quad fv(\phi)\cap fv(\psi)=\emptyset}{\langle P\|Q,\,\phi\wedge\psi\rangle \xrightarrow{\overline{\delta^c_{id_Q}}(t)} \langle P'\|Q',\,\varphi\rangle \xrightarrow{\delta^c_{id_P}(t)} \langle P'\|Q',\,\varphi\rangle}$$

`Restriction` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\alpha} \langle P',\,\phi'\rangle \quad \text{and} \quad \phi^L_\alpha \not\Leftrightarrow \mathbf{0}}{\langle P\backslash L,\,\phi\rangle \xrightarrow{\alpha} \langle P'\backslash L,\,\phi'\wedge\phi^L_\alpha\rangle}$$

`Observation` $$\frac{\langle P,\,\phi\rangle \xrightarrow{\gamma} \langle P',\,\phi'\rangle \quad \text{and} \quad \gamma\in\mathcal{O}^{-1}(\alpha)}{\langle P/\mathcal{O},\,\phi\rangle \xrightarrow{\alpha} \langle P'/\mathcal{O},\,\phi'\rangle}$$

Figure 1: *Semantics of constrained processes.*

In rule `Match`, we have

$$\psi \ ::= \begin{cases} \exists_x(\phi \wedge t == t') & \text{if } \alpha = \ c(x) \\ \exists_x(\phi \wedge t == t') \wedge \phi_f(t) \wedge x == f(t) & \text{if } \alpha = \ x := f(t) \\ \exists_x(\phi \wedge t == t') \wedge \phi_{new}(x) & \text{if } \alpha = \ x := new(-) \\ \phi' \wedge t == t' & \text{otherwise.} \end{cases}$$

The `Sum`, `Parallel`, `Protocol` and `Synchronisation` rules are assumed to be both associative and commutative. Moreover, constrained processes $\langle P,\,\phi\rangle$ and $\langle Q,\,\psi\rangle$ are defined with different variables, thus $fv(\phi)\cap fv(\psi)=\emptyset$. In `Synchronisation`, we have $\varphi ::= \phi' \wedge \psi' \wedge x == t$. In `Restriction`, ϕ^L_α is a formula such that $fv(\phi^L_\alpha) = fv(\alpha)$ and, for every valuation ϱ,

$$\varrho \models \phi^L_\alpha \quad \text{iff} \quad \varrho(\alpha) \in Act \setminus L\,.$$

Moreover, we need to restrict the `Restriction` semantics rule to the sets L such that every formula ϕ_α^L is definable within our logic. In `Observation`, the *computation* $\langle P,\ \phi\rangle \xrightarrow{\gamma} \langle P',\ \phi'\rangle$, for a sequence of actions $\gamma = \alpha_0\alpha_1 \ldots \alpha_n \in Act^*$, stands for the finite string of transitions satisfying

$$\langle P,\ \phi\rangle \xrightarrow{\alpha_0} \langle P_1,\ \phi_1\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \langle P',\ \phi'\rangle.$$

In the operational semantics rules for constrained processes, we assume that any transition $\langle P,\ \phi\rangle \xrightarrow{\alpha} \langle P',\ \phi'\rangle$ is dismissed whenever either ϕ or ϕ' is equivalent to $\mathbf{0}$.

A constrained process $\langle P',\ \phi'\rangle$ is a *derivative* of $\langle P,\ \phi\rangle$ if there is a computation $\langle P,\ \phi\rangle \xrightarrow{\gamma} \langle P',\ \phi'\rangle$ for some $\gamma \in Act^*$. We shall frequently use the set of $\langle P,\ \phi\rangle$'s derivatives defined by

$$\mathcal{D}(\langle P,\ \phi\rangle) = \{\langle P',\ \phi'\rangle \mid \exists_{\gamma \in Act^*}\ \langle P,\ \phi\rangle \xrightarrow{\gamma} \langle P',\ \phi'\rangle\}.$$

Example 3.1 *Consider the following SPPA processes:*

$$P ::=\ c(x_1).P_1, \quad P_1 ::=\ c(x_2).P_2 \quad \textit{and} \quad P_2 ::=\ [x_1 = x_2]\,\overline{c}(x_1).P_1\ .$$

The semantics of the constrained process $\langle P,\ \mathbf{1}\rangle$ is illustrated in Fig. 2.

$$\langle P,\ \mathbf{1}\rangle \xrightarrow{c(x_1)} \langle P_1,\ \mathcal{M}(x_1)\rangle \xrightarrow{c(x_2)} \langle P_2,\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle \ \underset{c(x_2)}{\overset{\overline{c}(x_1)}{\rightleftarrows}} \langle P_1,\ x_1 == x_2\rangle$$

Figure 2: Symbolic Semantics of $\langle P,\ \mathbf{1}\rangle$.

Example 3.2 *Consider the process*

$$Q ::=\ c(y_2).Q_1, \quad Q_1 ::=\ enc_{id_B}(y_1, y_2 : y_3).Q_2 \quad \textit{and} \quad Q_2 ::= \overline{c}(y_3).\mathbf{0}$$

where $fv(Q) = \{y_1\}$. The symbolic semantics of the constrained process $\langle Q,\ \mathcal{M}(y_1)\rangle$ is given in Fig. 3, where $\phi ::= \mathcal{K}(y_1) \wedge \mathcal{M}(y_2) \wedge y_3 == \{y_2\}_{y_1}$.

$$\langle Q,\ \mathcal{M}(y_1)\rangle \xrightarrow{c(y_2)} \langle Q_1,\ \mathcal{M}(y_1) \wedge \mathcal{M}(y_2)\rangle \xrightarrow{y_3 := enc_{id}(y_1, y_2)} \langle Q_2,\ \phi\rangle \xrightarrow{\overline{c}(y_3)} \langle \mathbf{0},\ \phi\rangle$$

Figure 3: Symbolic Semantics of $\langle Q,\ \mathcal{M}(y_1)\rangle$.

3.3 Symbolic vs Value-Passing Semantics

The relationship between the symbolic operational semantics of constrained processes and the value-passing operational semantics of processes (see Appendix A) is detailed in the following lemmas. Every sequence of transitions between SPPA processes can be unwinded to a sequence of transitions between constrained processes. Conversely, every transition between constrained processes can be interpreted as a set of transitions between processes.

Lemma 3.3 *Let P, P' be SPPA processes, let $\alpha' = \overline{c}(t), \delta_{id}^c(t), \overline{\delta_{id}^c}(t)$ or τ, and let $\beta' = c(x)$ or $x := f(t)$, for some term t such that $fv(t) \subseteq \{x_1, \ldots, x_n\}$.*

- *If $P[a_1/x_1] \ldots [a_n/x_n] \xrightarrow{\alpha} P'[a_1/x_1] \ldots [a_n/x_n]$, for some $a_1, \ldots, a_n \in \mathcal{M}$ ($n \geq 0$) and $\alpha = \alpha'[a_1/x_1] \ldots [a_n/x_n]$, then, for any formula $\phi \not\Leftrightarrow \mathbf{0}$ such that $fv(\phi) = \{x_1, \ldots, x_n\}$, we have*

$$\langle P,\ \phi\rangle \xrightarrow{\alpha'} \langle P',\ \phi'\rangle$$

where ϕ' is the formula given by the symbolic operational semantics ($\phi' \not\Leftrightarrow \mathbf{0}$).

- *If* $P[a_1/x_1]\dots[a_n/x_n] \xrightarrow{\beta} P'[a_1/x_1]\dots[a_n/x_n][a/x]$, *for some* $a_1,\dots,a_n,a \in \mathcal{M}$ $(n \geq 0)$ *and* $\beta = \beta'[a_1/x_1]\dots[a_n/x_n][a/x]$, *then, for any formula* ϕ *such that* $fv(\phi) = \{x_1,\dots,x_n\}$, *we have*

$$\langle P,\ \phi\rangle \xrightarrow{\beta'} \langle P',\ \phi'\rangle$$

where ϕ' *is the formula given by the symbolic operational semantics* ($\phi' \not\Leftrightarrow \mathbf{0}$).

Lemma 3.4 *Let* P, P' *be SPPA processes, let* $\alpha' = \overline{c}(t), \delta^c_{id}(t), \overline{\delta^c_{id}}(t)$ *or* τ, *and let* $\beta' = c(x)$ *or* $x := f(t)$, *for some term* t *such that* $fv(t) \subseteq \{x_1,\dots,x_n\}$.

- *If* $\langle P,\ \phi\rangle \xrightarrow{\alpha'} \langle P',\ \phi'\rangle$, *with* $fv(\phi) = fv(\phi') = \{x_1,\dots,x_n\}$, *then, for every valuation* ϱ *such that* $\varrho \models \phi'$,

$$P[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n] \xrightarrow{\alpha} P'[\varrho(x_1)/x_1]\dots[\varrho(x_1)/x_n]$$

where $\alpha = \varrho(\alpha')$.

- *If* $\langle P,\ \phi\rangle \xrightarrow{\beta'} \langle P',\ \phi'\rangle$, *with* $fv(\phi) = \{x_1,\dots,x_n\}$ *and* $fv(\phi') = \{x_1,\dots,x_n,x\}$, *then, for every valuation* ϱ *such that* $\varrho \models \phi'$,

$$P[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n] \xrightarrow{\beta} P'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n][\varrho(x)/x]$$

where $\beta = \varrho(\beta')$.

Lemma 3.5 *Let* P, P' *be SPPA processes, let* $\alpha' = \overline{c}(t), \delta^c_{id}(t), \overline{\delta^c_{id}}(t)$ *or* τ, *and let* $\beta' = c(x)$ *or* $x := f(t)$, *for some term* t *such that* $fv(t) \subseteq \{x_1,\dots,x_n\}$. *Consider* $a_1,\dots,a_n,a \in \mathcal{M}$ *for some* $n \geq 0$, *and let* $\alpha = \alpha'[a_1/x_1]\dots[a_n/x_n]$ *and* $\beta = \beta'[a_1/x_1]\dots[a_n/x_n][a/x]$.

- *If* $P[a_1/x_1]\dots[a_n/x_n] \xrightarrow{\alpha} P'[a_1/x_1]\dots[a_n/x_n]$ *and* $\models \phi[a_1/x_1]\dots[a_n/x_n]$, *then* $\models \phi'[a_1/x_1]\dots[a_n/x_n]$ *whenever*

$$\langle P,\ \phi\rangle \xrightarrow{\alpha'} \langle P',\ \phi'\rangle.$$

- *If* $P[a_1/x_1]\dots[a_n/x_n] \xrightarrow{\beta} P'[a_1/x_1]\dots[a_n/x_n][a/x]$ *and* $\models \phi[a_1/x_1]\dots[a_n/x_n]$, *then* $\models \phi'[a_1/x_1]\dots[a_n/x_n][a/x]$ *whenever*

$$\langle P,\ \phi\rangle \xrightarrow{\beta'} \langle P',\ \phi'\rangle.$$

The proofs of Lemma 3.3, Lemma 3.4 and Lemma 3.5 are given in Appendix C.

4 Bisimulation Equivalence for Constrained Processes

In this section, we extend Milner's notion of strong bisimulation [Mil89] to handle constrained processes. But first, in order to bind the variables of the compared constrained processes, we need consider finite relations between their free variables. Recall that whenever we compare two processes P and Q, we always assume that no variable has been used in both definition.

Let $\mathcal{V}_1, \mathcal{V}_2 \subseteq \mathcal{V}$ be a finite subset of variables and let $R \subseteq \mathcal{V}_1 \times \mathcal{V}_2$ be a relation between $\mathcal{V}_1$ and $\mathcal{V}_2$ such that $\forall_{x\in\mathcal{V}_1}\exists_{y\in\mathcal{V}_2}\ (x,y) \in R$ and $\forall_{y\in\mathcal{V}_2}\exists_{x\in\mathcal{V}_1}\ (x,y) \in R$. (Note that we must have $R = \emptyset$ whenever $\mathcal{V}_1 = \emptyset$ or $\mathcal{V}_2 = \emptyset$.) For any $x \in \mathcal{V}$, we use $R[x]$ to denote the relation between $\mathcal{V}_1 \setminus \{x\}$ and $\mathcal{V}_2$ such that $R[x] = \{(x',y') \in R \mid x' \neq x \text{ and } y' \neq x\}$. In addition, for any $x, y \in \mathcal{V}$, we use $R[(x,y)]$ to denote the relation between $\mathcal{V}_1 \cup \{x\}$ and $\mathcal{V}_2 \cup \{y\}$ defined as

$$R[(x,y)] = R[x][y]\ \cup\ \{(x,y)\}.$$

Therefore, $R[(x,y)]$ is obtained from R by, first removing every occurrence of x and y, and then adding (x,y). The set of all relations between finite subsets of variables is denoted by $\mathcal{FR}$.

A valuation ϱ is said to be *consistent* with the relation R if $\varrho(x) = \varrho(y)$ whenever $(x,y) \in R$. Given $x, y \in \mathcal{V}$, $\varrho[x/y]$ denotes the valuation obtained from ϱ by setting $\varrho[x/y](y) = \varrho(x)$ (and $\varrho[x/y](z) = \varrho(z)$ otherwise). It is easy to see that the valuation $\varrho[x/y]$ is consistent with the relation $R[(x,y)]$ whenever ϱ is consistent with $R[x]$.

Definition 4.1 (Bisimulation) *Let $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$ be constrained processes and let R be a relation between $fv(\phi)$ and $fv(\psi)$. A* bisimulation *between $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$ with respect to R is a family of relations $\mathcal{R} = \{\mathcal{R}^{\varrho}\}_{\varrho}$, for every valuation ϱ, where each relation $\mathcal{R}^{\varrho} \subseteq \mathcal{D}(\langle P,\ \phi\rangle) \times \mathcal{D}(\langle Q,\ \psi\rangle) \times \mathcal{FR}$ satisfies the following conditions:*

1. *If $\varrho \models \phi$, $\varrho \models \psi$ and ϱ is consistent with R, then $(\langle P,\ \phi\rangle, \langle Q,\ \psi\rangle, R) \in \mathcal{R}^{\varrho}$;*
2. *whenever $(\langle P_1,\ \phi_1\rangle, \langle Q_1,\ \psi_1\rangle, R_1) \in \mathcal{R}^{\varrho}$, for any actions $\alpha \in \{\overline{c}(t),\ \delta^c_{id}(t),\ \overline{\delta^c_{id}}(t),\ \tau \mid t \in \mathcal{T},\ c \in C\}$ and $\beta \in \{c(x),\ x := f(t),\ x := new(-) \mid t \in \mathcal{T},\ c \in C\}$, we have*
   - *if $\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha} \langle P_2,\ \phi_2\rangle$ and $\varrho \models \phi_2$, then there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha'} \langle Q_2,\ \psi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1) \in \mathcal{R}^{\varrho}$ and $\varrho \models \psi_2$;*
   - *if $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha} \langle Q_2,\ \psi_2\rangle$ and $\varrho \models \psi_2$, then there is a transition $\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha'} \langle P_2,\ \phi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(y_i, x_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1) \in \mathcal{R}^{\varrho}$ and $\varrho \models \phi_2$;*
   - *if $\langle P_1,\ \phi_1\rangle \xrightarrow{\beta} \langle P_2,\ \phi_2\rangle$, then for every valuation ϱ' consistent with $R_1[x]$ such that $\varrho' \models \phi_2$, there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta'} \langle Q_2,\ \psi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(x, y)]) \in \mathcal{R}^{\varrho'[x/y]}$ and $\varrho'[x/y] \models \psi_2$;*
   - *if $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta} \langle Q_2,\ \psi_2\rangle$, then for every valuation ϱ' consistent with $R_1[x]$ such that $\varrho' \models \phi_2$, there is a transition $\langle P_1,\ \phi_1\rangle \xrightarrow{\beta'} \langle P_2,\ \phi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(y_i, x_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(y, x)]) \in \mathcal{R}^{\varrho'[x/y]}$ and $\varrho'[x/y] \models \phi_2$.*

Two constrained processes $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$ are bisimilar *if there exists a bisimulation which relates $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$ with respect to some relation R between $fv(\phi)$ and $fv(\psi)$. In that case, we write $\langle P,\ \phi\rangle \simeq \langle Q,\ \psi\rangle$.*

Note that if $\mathcal{R} = \{\mathcal{R}^{\varrho}\}_{\varrho}$ is a bisimulation, $\varrho \models \phi$ and $\varrho \models \phi$ whenever $(\langle P,\ \phi\rangle, \langle Q,\ \psi\rangle, R) \in \mathcal{R}^{\varrho}$. Moreover, ϱ must be consistent with the relation R.

4.1 Equivalence of the Bisimulations

In the following theorem, we see that the bisimulation equivalence between SPPA constrained processes, as defined above, corresponds to the strong bisimulation equivalence $\simeq$ between processes. Obviously, this result holds only when the SPPA processes under comparison are compatible i.e., for processes without free variable and generating prefix. (We say that an SPPA process P is *closed* whenever $fv(P) = \emptyset$.) First, we define strong bisimulation between value-passing processes.

Definition 4.2 *A bisimulation between closed processes P and Q is a relation $\mathcal{R} \subseteq \mathcal{D}(P) \times \mathcal{D}(Q)$ such that*

- *$(P, Q) \in \mathcal{R}$;*
- *If $(P_1, Q_1) \in \mathcal{R}$ and $P_1 \xrightarrow{\alpha} P_2$, then there is a transition $Q_1 \xrightarrow{\alpha} Q_2$ such that $(P_2, Q_2) \in \mathcal{R}$;*
- *If $(P_1, Q_1) \in R$ and $Q_1 \xrightarrow{\alpha} Q_2$, then there is a transition $P_1 \xrightarrow{\alpha} P_2$ such that $(P_2, Q_2) \in \mathcal{R}$;*

If there exists a bisimulation which relates P and Q, then we write $P \simeq Q$.

Theorem 4.3 *Let P and Q be closed processes without generating prefixes. Then, $P \simeq Q$ if and only if $\langle P,\ \mathbf{1}\rangle \simeq \langle Q,\ \mathbf{1}\rangle$.*

Proof: ($\Longrightarrow$) Let $\mathcal{R}$ be a bisimulation between P and Q. Consider the family of relations $\mathcal{R}' = \{\mathcal{R}'^{\varrho}\}_{\varrho}$, where, given a valuation ϱ the relation $\mathcal{R}'^{\varrho}$ is defined as follows:

- for every $(P', Q') \in \mathcal{R}$, $(\langle P',\ \mathbf{1}\rangle, \langle Q',\ \mathbf{1}\rangle, \emptyset) \in \mathcal{R}'^{\varrho}$;
- for every $(P'[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n], Q'[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]) \in \mathcal{R}$, for every formulas ϕ and ψ, and for every relation R between $fv(\phi) = \{x_1, \ldots, x_n\}$ and $fv(\psi) = \{y_1, \ldots, y_m\}$, $(\langle P',\ \phi\rangle, \langle Q',\ \psi\rangle, R) \in \mathcal{R}'^{\varrho}$ whenever $\varrho \models \phi$, $\varrho \models \psi$ and ϱ is consistent with R.

In the following, we show that $\mathcal{R}' = \{\mathcal{R}'^{\varrho}\}_{\varrho}$ is a bisimulation between $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$ with respect to the relation $\emptyset$. For that purpose, we show that each relation $\mathcal{R}'^{\varrho}$ satisfies the conditions from Definition 4.1.

1. First, we see that $(\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset) \in \mathcal{R}'^{\varrho}$ (with $\varrho \models \mathbf{1}$ and ϱ (always) consistent with $\emptyset$).

2. Let $(\langle P_1,\ \phi_1\rangle, \langle Q_1,\ \psi_1\rangle, R) \in \mathcal{R}'^{\varrho}$. First, we see that $\varrho \models \phi_1$, $\varrho \models \psi_1$ and ϱ is consistent with R. Moreover, assume that $fv(\phi_1) = \{x_1, \ldots, x_n\}$ and $fv(\psi_1) = \{y_1, \ldots, y_m\}$, with R a relation between $fv(\phi_1)$ and $fv(\psi_1)$. Also, we have $(P_1', Q_1') \in \mathcal{R}$, where $P_1' ::= P_1[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n]$ and $Q_1' ::= Q_1[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]$.

   Assume that $\varrho \models \phi_2$ and $\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha'} \langle P_2,\ \phi_2\rangle$ with $\alpha' ::= \overline{c}(t),\ \delta^c_{id}(t), \overline{\delta^c_{id}}(t)$ or τ. By Lemma 3.4,

   $$P_1' \xrightarrow{\alpha} P_2'$$

   where $P_2' ::= P_2[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n]$ and $\alpha = \varrho(\alpha')$. Since $(P_1', Q_1') \in \mathcal{R}$, there is a transition

   $$Q_1' \xrightarrow{\alpha} Q_2'$$

   such that $(P_2', Q_2') \in \mathcal{R}$, where $Q_2' ::= Q_2[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]$. By Lemma 3.3, there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha''} \langle Q_2,\ \psi_2\rangle$ with $\alpha = \alpha''[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]$ and $\varrho \models \psi_2$ (by Lemma 3.5). Moreover, since ϱ is consistent with R, we may assume that $\alpha' = \alpha''[y_1'/x_1]\ldots[y_n'/x_n]$ with $(x_i, y_i') \in R$ and $\{y_1', \ldots, y_n'\} = \{y_1, \ldots, y_m\}$. Therefore, since $\varrho \models \phi_2$, $\varrho \models \psi_2$ and ϱ is consistent with R, we see that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R) \in \mathcal{R}'^{\varrho}$. The case where $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha'} \langle Q_2,\ \psi_2\rangle$ is similar.

   Now assume that $\langle P_1,\ \phi_1\rangle \xrightarrow{\beta'} \langle P_2,\ \phi_2\rangle$, with $\beta ::= c(x)$ or $x := f(t)$, and let ϱ' be a valuation consistent with $R[x]$ such that $\varrho' \models \phi_2$. By Lemma 3.4,

   $$P_1' \xrightarrow{\beta} P_2'$$

   where $\beta = \varrho'(\beta')$ and $P_2' ::= P_2[\varrho'(x_1)/x_1]\ldots[\varrho'(x_n)/x_n][\varrho'(x)/x]$. Since $(P_1', Q_1') \in \mathcal{R}$, there is a transition

   $$Q_1' \xrightarrow{\beta} Q_2'$$

   such that $(P_2', Q_2') \in \mathcal{R}$, where $Q_2' ::= Q_2[\varrho'(y_1)/y_1]\ldots[\varrho'(y_m)/y_m][\varrho'(x)/y]$ for some y. By Lemma 3.3 there is a transition

   $$\langle Q_1,\ \phi_1\rangle \xrightarrow{\beta''} \langle Q_2,\ \psi_2\rangle$$

   with $\beta = \beta''[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m][\varrho'(x)/y] = \varrho'(\beta'')$ and $\varrho'[x/y] \models \psi_2$ (by Lemma 3.5). Moreover, since ϱ' is consistent with $R[x]$, we may assume that $\beta' = \beta''[y_1'/x_1]\ldots[y_n'/x_n][y/x]$ with $(x_i, y_i') \in R$ and $\{y_1', \ldots, y_n'\} = \{y_1, \ldots, y_m\}$. Therefore, since $\varrho'[x/y] \models \phi_2$, $\varrho'[x/y] \models \psi_2$ and $\varrho'[x/y]$ is consistent with $R[(x,y)]$, we see that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R[(x,y)]) \in \mathcal{R}'^{\varrho'[x/y]}$. The case where $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta'} \langle Q_2,\ \psi_2\rangle$ is similar.

($\Longleftarrow$) Let $\mathcal{R} = \{\mathcal{R}^{\varrho}\}$ be a bisimulation between $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$ with respect to $\emptyset$. Consider the relation $\mathcal{R}' \subseteq \mathcal{D}(P) \times \mathcal{D}(Q)$ defined as follows:

- If $(\langle P',\ \mathbf{1}\rangle,\ \langle Q',\ \mathbf{1}\rangle,\ \emptyset) \in \mathcal{R}^{\varrho}$ for some ϱ, then $(P', Q') \in \mathcal{R}'$;
- If $(\langle P',\ \phi\rangle,\ \langle Q',\ \psi\rangle,\ R) \in \mathcal{R}^{\varrho}$ for some ϱ, then

  $$(P'[\varrho(x_1)/x_1]\ldots[\varrho(x_1)/x_n],\ Q'[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]) \in \mathcal{R}'$$

  where $fv(\phi) = \{x_1, \ldots, x_n\}$ and $fv(\psi) = \{y_1, \ldots, y_m\}$.

In the following, we show that $\mathcal{R}'$ is a bisimulation between P and Q.

1. First, we see that $(P, Q) \in \mathcal{R}'$ since $(\langle P,\ \mathbf{1}\rangle,\ \langle Q,\ \mathbf{1}\rangle, \emptyset) \in \mathcal{R}^{\varrho}$. for every valuation ϱ.

2. Let $(P_1[a_1/x_1]\ldots[a_n/x_n], /Q_1[b_1/y_1]\ldots[b_m/y_m]) \in \mathcal{R}'$ where $a_i, b_j \in \mathcal{M}$. By the definition of $\mathcal{R}'$, there is a valuation ϱ, with $\varrho(x_i) = a_i$ and $\varrho(y_j) = b_j$, such that $(\langle P_1,\ \phi_1\rangle,\ \langle Q_1,\ \psi_1\rangle,\ R) \in \mathcal{R}^{\varrho}$ with $fv(\phi_1) = \{x_1, \ldots, x_n\}$ and $fv(\psi_1) = \{y_1, \ldots, y_m\}$. Put $P_1' ::= P_1[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n]$ and $Q_1' ::= Q_1[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]$. Moreover, we see that $\varrho \models \phi_1$, $\varrho \models \psi_1$ and ϱ is consistent with R.

   Let $\alpha' ::= \overline{c}(t),\ \delta^c_{id}(t),\ \overline{\delta^c_{id}}(t)$ or τ, and assume that $P_1' \xrightarrow{\alpha} P_2'$ where $\alpha = \alpha'[a_1/x_1]\ldots[a_n/x_n] = \alpha'[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n] = \varrho(\alpha)$. and $P_2' ::= P_2[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n]$. By Lemma 3.3 and since $fv(\phi_1) = \{x_1, \ldots, x_n\}$, we have

   $$\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha'} \langle P_2,\ \phi_2\rangle$$

   with $\varrho \models \phi_2$ (by Lemma 3.5). But since $(\langle P_1,\ \phi_1\rangle,\ \langle Q_1,\ \psi_1\rangle,\ R) \in \mathcal{R}^{\varrho}$, there is a transition

   $$\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha''} \langle Q_2,\ \psi_2\rangle$$

for $\alpha' = \alpha[y'_1/x_1]\ldots[y'_n/x_n]$ with $\{y'_1, \ldots, y'_n\} = \{y_1, \ldots, y_m\}$ and $(x_i, y'_i) \in R$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R) \in \mathcal{R}^{\varrho}$ and $\varrho \models \psi_2$. Also, since ϱ is consistent with R, we have $\varrho(\alpha'') = \varrho(\alpha') = \alpha$. Therefore, by Lemma 3.4 and since $\varrho \models \psi_2$, we have

$$Q'_1 \xrightarrow{\alpha} Q'_2$$

where $Q'_2 ::= Q_2[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m]$. Moreover, since $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R) \in \mathcal{R}^{\varrho}$ and since $fv(\phi_2) = fv(\phi_1) = \{x_1, \ldots, x_n\}$ and $fv(\psi_2) = fv(\psi_1) = \{y_1, \ldots, y_m\}$, we see from the definition of $\mathcal{R}'$ that $(P'_2, Q'_2) \in \mathcal{R}'$. The case where $Q'_1 \xrightarrow{\alpha} Q'_2$ is similar.

Now assume that $P'_1 \xrightarrow{\beta} P'_2$ where $P'_2 ::= P_2[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n][a/x]$ and $\beta = \beta'[a_1/x_1]\ldots[a_n/x_n][a/x] = \beta'[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n][a/x]$ for some $\beta' ::= c(x)$ or $x := f(t)$, Consider valuation ϱ' defined as follows:

$$\varrho'(x) = a \quad \text{and} \quad \varrho'(z) = \varrho(z) \text{ otherwise.}$$

Then ϱ' is consistent with $R[x]$ and $\beta = \varrho'(\beta')$. Futhermore, we see that $P'_2 = P_2[\varrho'(x_1)/x_1]\ldots[\varrho'(x_n)/x_n][\varrho'(x)/x]$. By Lemma 3.3, and since $fv(\phi_1) = \{x_1, \ldots, x_n\}$, we have

$$\langle P_1,\ \phi_1\rangle \xrightarrow{\beta'} \langle P_2,\ \phi_2\rangle$$

with $\varrho' \models \phi_2$ (by Lemma 3.5). But since $(\langle P_1,\ \phi_1\rangle,\ \langle Q_1,\ \psi_1\rangle,\ R) \in \mathcal{R}^{\varrho}$, there is a transition

$$\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta''} \langle Q_2,\ \psi_2\rangle$$

for $\beta' = \beta[y'_1/x_1]\ldots[y'_n/x_n]$ with $\{y'_1, \ldots, y'_n\} = \{y_1, \ldots, y_m\}$ and $(x_i, y'_i) \in R$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R[(x, y)]) \in \mathcal{R}^{\varrho'[x/y]}$ and $\varrho'[x/y] \models \psi_2$. Also, since ϱ' is consistent with $R[x]$, therefore ϱ' is consistent with $R[(x, y)]$, we have $\varrho'[x/y](\beta'') = \varrho'[x/y](\beta') = \beta$. Hence, by Lemma 3.4 and since $\varrho'[x/y] \models \psi_2$, we have

$$Q'_1 \xrightarrow{\beta} Q'_2$$

where $Q'_2 ::= Q_2[\varrho'[x/y](y_1)/y_1]\ldots[\varrho'[x/y](y_m)/y_m][\varrho'[x/y](y)/y] = Q_2[\varrho(y_1)/y_1]\ldots[\varrho(y_m)/y_m][a/y]$. Moreover, since $fv(\phi_2) = fv(\phi_1) \cup \{x\} = \{x_1, \ldots, x_n, x\}$ and $fv(\psi_2) = fv(\psi_1) \cup \{y\} = \{y_1, \ldots, y_m, y\}$ and since $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R[(x, y)]) \in \mathcal{R}^{\varrho'[x/y]}$, we see from the definition of $\mathcal{R}'$ that $(P'_2, Q'_2) \in \mathcal{R}'$. The case where $Q'_1 \xrightarrow{\beta} Q'_2$ is similar.

□

4.2 Symbolic Bisimulation

In the following, we consider constrained processes $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$. Consider the following sets:

$$\{\phi' \mid \langle P',\ \phi'\rangle \in \mathcal{D}(\langle P,\ \phi\rangle) \text{ for some } P'\} = \{\phi_1, \ldots, \phi_m\},$$

$$\{\psi' \mid \langle Q',\ \psi'\rangle \in \mathcal{D}(\langle Q,\ \psi\rangle) \text{ for some } Q'\} = \{\psi_1, \ldots, \psi_{m'}\},$$

$$\bigcup_{1\le j\le m} fv(\phi_j) = \{x_1, \ldots, x_n\} \quad \text{and} \quad \bigcup_{1\le j\le m'} fv(\psi_j) = \{y_1, \ldots, y_{n'}\}.$$

Such sets of formulas and variables are finite since the sets $\mathcal{D}(\langle P,\ \phi\rangle)$ and $\mathcal{D}(\langle Q,\ \psi\rangle)$ are finite (recall that $\langle P,\ \phi\rangle$ and $\langle P,\ \phi'\rangle$ are identical whenever $\phi \Leftrightarrow \phi'$ and $fv(\phi) = fv(\phi')$, thus we may assume that the ϕ_i (resp. the ψ_i) are pairwise not equivalent). Now consider the equivalence relation over valuations defined as follows: $\varrho \equiv \varrho'$ if

1. for every $1 \le i \le m$, $\quad \varrho \models \phi_i \quad$ iff $\quad \varrho' \models \phi_i$;
2. for every $1 \le i \le m'$, $\quad \varrho \models \psi_i \quad$ iff $\quad \varrho' \models \psi_i$;
3. for every $z, z' \in \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_{n'}\}$,

$$\varrho(z) = \varrho(z') \quad \text{iff} \quad \varrho'(z) = \varrho'(z').$$

Also consider the equivalence class (with respect to $\langle P,\ \phi\rangle$ and $\langle Q,\ \psi\rangle$) of a valuation ϱ defined as follows:

$$[\![\varrho]\!] = \{\varrho' \mid \varrho' \equiv \varrho\}.$$

Lemma 4.4 *Let $\varrho \equiv \varrho'$.*

1. *For any relation R between a subset of $\{x_1, \ldots, x_n\}$ and a subset of $\{y_1, \ldots, y_{n'}\}$,*

$$\varrho \text{ is consistent with } R \text{ if and only if } \varrho' \text{ is consistent with } R.$$

2. *For any $x, y \in \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_{n'}\}$, $\varrho[x/y] \equiv \varrho'[x/y]$.*

The proof of Lemma 4.4 is straightforward from the definition of the equivalence relation $\equiv$.

Lemma 4.5 *There are finitely many equivalence classes $[\![\varrho]\!]$ i.e., the set $\{[\![\varrho]\!] \mid \varrho \text{ is a valuation}\}$ is finite.*

Proof: Follows from the fact that the sets $\{\phi' \mid \langle P', \phi'\rangle \in \mathcal{D}(\langle P, \phi\rangle)$ for some $P'\}$ and $\{\psi' \mid \langle Q', \psi'\rangle \in \mathcal{D}(\langle Q, \psi\rangle)$ for some $Q'\}$ are finite. Therefore, using the notation established above, there are at most $2^{m+m'+(n+n'-1)(n+n')/2}$ equivalence classes $[\![\varrho]\!]$. □

Definition 4.6 (Symbolic Bisimulation) *Let $\langle P, \phi\rangle$ and $\langle Q, \psi\rangle$ be constrained processes and let R be a relation between $fv(\phi)$ and $fv(\psi)$. A* symbolic bisimulation *between $\langle P, \phi\rangle$ and $\langle Q, \psi\rangle$ with respect to R is a finite family $\mathcal{R} = \{\mathcal{R}^{[\![\varrho]\!]}\}_\varrho$, for every valuation ϱ, where each relation $\mathcal{R}^{[\![\varrho]\!]}$ satisfies the conditions:*

1. *If $\varrho \models \phi$, $\varrho \models \psi$ and ϱ is consistent with R, then $(\langle P, \phi\rangle, \langle Q, \psi\rangle, R) \in \mathcal{R}^{[\![\varrho]\!]}$;*
2. *Whenever $(\langle P_1, \phi_1\rangle, \langle Q_1, \psi_1\rangle, R_1) \in \mathcal{R}^{[\![\varrho]\!]}$, for any actions $\alpha \in \{\overline{c}(t), \delta^c_{id}(t), \overline{\delta^c_{id}}(t), \tau \mid t \in \mathcal{T}, c \in C\}$ and $\beta \in \{c(x), x := f(t), x := new(-) \mid t \in \mathcal{T}, c \in C\}$, we have*
   - *if $\langle P_1, \phi_1\rangle \xrightarrow{\alpha} \langle P_2, \phi_2\rangle$ and $\varrho \models \phi_2$, then there is a transition $\langle Q_1, \psi_1\rangle \xrightarrow{\alpha'} \langle Q_2, \psi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1) \in \mathcal{R}^{[\![\varrho]\!]}$ and $\varrho \models \psi_2$;*
   - *if $\langle Q_1, \psi_1\rangle \xrightarrow{\alpha} \langle Q_2, \psi_2\rangle$ and $\varrho \models \psi_2$, then there is a transition $\langle P_1, \phi_1\rangle \xrightarrow{\alpha'} \langle P_2, \phi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(y_i, x_i) \in R_1$, such that $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1) \in \mathcal{R}^{[\![\varrho]\!]}$ and $\varrho \models \phi_2$;*
   - *if $\langle P_1, \phi_1\rangle \xrightarrow{\beta} \langle P_2, \phi_2\rangle$, then for every valuation ϱ' consistent with $R_1[x]$ such that $\varrho' \models \phi_2$, there is a transition $\langle Q_1, \psi_1\rangle \xrightarrow{\beta'} \langle Q_2, \psi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1[(x, y)]) \in \mathcal{R}^{[\![\varrho'[x/y]]\!]}$ and $\varrho'[x/y] \models \psi_2$;*
   - *if $\langle Q_1, \psi_1\rangle \xrightarrow{\beta} \langle Q_2, \psi_2\rangle$, then for every valuation ϱ' consistent with $R_1[x]$ such that $\varrho' \models \phi_2$, there is a transition $\langle P_1, \phi_1\rangle \xrightarrow{\beta'} \langle P_2, \phi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(y_i, x_i) \in R_1$, such that $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1[(y, x)]) \in \mathcal{R}^{[\![\varrho'[x/y]]\!]}$ and $\varrho'[x/y] \models \phi_2$.*

We write $\langle P, \phi\rangle \simeq_s \langle Q, \psi\rangle$ whenever there exists a symbolic bisimulation which relates $\langle P, \phi\rangle$ and $\langle Q, \psi\rangle$ with respect to some relation R between $fv(\phi)$ and $fv(\psi)$.

The following theorem states that symbolic bisimulation is a sound and complete method for verifying bisimilarity between constrained processes.

Theorem 4.7 *$\langle P, \phi\rangle \simeq \langle Q, \psi\rangle$ if and only if $\langle P, \phi\rangle \simeq_s \langle Q, \psi\rangle$.*

Proof: First, assume that $\langle P, \phi\rangle \simeq \langle Q, \psi\rangle$, and let $\mathcal{R} = \{\mathcal{R}^\varrho\}_\varrho$ be a bisimulation with respect to some relation R between $fv(\phi)$ and $fv(\psi)$. For every equivalent class $[\![\varrho']\!]$, choose one valuation $\varrho \in [\![\varrho']\!]$ and define $\mathcal{R}'^{[\![\varrho']\!]} = \mathcal{R}^\varrho$. Then it is enough to show that the (finite) family $\mathcal{R}' = \{\mathcal{R}'^{[\![\varrho]\!]}\}_\varrho$ is a symbolic bisimulation with respect to R, therefore $\langle P, \phi\rangle \simeq_s \langle Q, \psi\rangle$. Indeed, given an equivalent class $[\![\varrho']\!]$, with $\mathcal{R}'^{[\![\varrho']\!]} = \mathcal{R}^\varrho$, we see that the $\mathcal{R}'^{[\![\varrho']\!]}$ satisfies every conditions from Definition 4.6.

1. If $\varrho' \models \phi$, then $\varrho \models \phi$; If $\varrho' \models \psi$, then $\varrho \models \psi$; and, by Lemma 4.4, if ϱ' is consistent with R, then ϱ is consistent with R. Therefore, $(\langle P, \phi\rangle, \langle Q, \psi\rangle, R) \in \mathcal{R}^\varrho$, thus $(\langle P, \phi\rangle, \langle Q, \psi\rangle, R) \in \mathcal{R}^{[\![\varrho']\!]}$.
2. Assume $(\langle P_1, \phi_1\rangle, \langle Q_1, \psi_1\rangle, R_1) \in \mathcal{R}^{[\![\varrho']\!]}$ with $\varrho' \models \phi_1$ and $\varrho' \models \psi_1$. Then, we have $(\langle P_1, \phi_1\rangle, \langle Q_1, \psi_1\rangle, R_1) \in \mathcal{R}^\varrho$ with $\varrho \models \phi_1$ and $\varrho \models \psi_1$. Therefore, if $\langle P_1, \phi_1\rangle \xrightarrow{\alpha} \langle P_2, \phi_2\rangle$ for $\alpha ::= \overline{c}(t), \delta^c_{id}(t), \overline{\delta^c_{id}}(t)$, or τ, and $\varrho' \models \phi_2$, thus $\varrho \models \phi_2$, then there is a transition $\langle Q_1, \psi_1\rangle \xrightarrow{\alpha'} \langle Q_2, \psi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1) \in \mathcal{R}^\varrho$ and $\varrho \models \psi_2$. Therefore $(\langle P_2, \phi_2\rangle, \langle Q_2, \psi_2\rangle, R_1) \in \mathcal{R}^{[\![\varrho']\!]}$ and $\varrho' \models \psi_2$. The case where $\langle Q_1, \psi_1\rangle \xrightarrow{\alpha} \langle Q_2, \psi_2\rangle$ is similar.

Now assume that $\langle P_1,\ \phi_1\rangle \xrightarrow{\beta} \langle P_2,\ \phi_2\rangle$, for $\beta ::= c(x),\ x := f(t)$ or $x := new(-)$, and let ϱ_1' be a valuation consistent with $R[x]$ and such that $\varrho_1' \models \phi_2$. Also, assume that $\mathcal{R}^{[\![\varrho_1']\!]} = \mathcal{R}^{\varrho_1}$. Then, by Lemma 4.4, the valuation ϱ_1 is consistent with $R[x]$ and $\varrho_1 \models \phi_2$ (since $\varrho_1 \equiv \varrho_1'$). Therefore, there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta'} \langle Q_2,\ \psi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(x,y)]) \in \mathcal{R}^{\varrho_1[x/y]}$ and $\varrho_1[x/y] \models \psi_2$. Thus, by Lemma 4.4, we have $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(x,y)]) \in \mathcal{R}^{[\![\varrho_1'[x/y]]\!]}$ and $\varrho_1'[x/y] \models \psi_2$. The case where $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta} \langle Q_2,\ \psi_2\rangle$ is similar.

Conversely, assume that $\langle P,\ \phi\rangle \simeq_{\mathrm{s}} \langle Q,\ \psi\rangle$, and let $\mathcal{R} = \{\mathcal{R}^{[\![\varrho]\!]}\}_\varrho$ be a symbolic bisimulation with respect to some relation R between $fv(\phi)$ and $fv(\psi)$.. Consider the family $\mathcal{R}' = \{\mathcal{R}^\varrho\}_\varrho$ where $\mathcal{R}^\varrho = \mathcal{R}^{[\![\varrho']\!]}$ whenever $\varrho \equiv \varrho'$. We see that $\mathcal{R}'$ is a bisimulation with respect to R, thus $\langle P,\ \phi\rangle \simeq \langle Q,\ \psi\rangle$. Indeed, given a valuation ϱ, we show that the relation $\mathcal{R}^\varrho = \mathcal{R}^{[\![\varrho']\!]}$ satisfies the conditions from Definition 4.1.

1. If $\varrho \models \phi$, then $\varrho' \models \phi$; If $\varrho \models \psi$, then $\varrho' \models \psi$; and, by Lemma 4.4, if ϱ is consistent with R, then ϱ' is consistent with R. Therefore, $(\langle P,\ \phi\rangle, \langle Q,\ \psi\rangle, R) \in \mathcal{R}^{[\![\varrho']\!]}$, thus $(\langle P,\ \phi\rangle, \langle Q,\ \psi\rangle, R) \in \mathcal{R}^\varrho$.

2. Assume $(\langle P_1,\ \phi_1\rangle, \langle Q_1,\ \psi_1\rangle, R_1) \in \mathcal{R}^\varrho$ with $\varrho \models \phi_1$ and $\varrho \models \psi_1$. Then, we have $(\langle P_1,\ \phi_1\rangle, \langle Q_1,\ \psi_1\rangle, R_1) \in \mathcal{R}^{lbbval']\!]}$ with $\varrho' \models \phi_1$ and $\varrho' \models \psi_1$. Therefore, if $\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha} \langle P_2,\ \phi_2\rangle$ for $\alpha ::= \overline{c}(t),\ \delta^c_{id}(t),\ \overline{\delta^c_{id}}(t)$, or τ, and $\varrho \models \phi_2$, thus $\varrho' \models \phi_2$, then there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha'} \langle Q_2,\ \psi_2\rangle$, for some $\alpha' = \alpha[y_1/x_1]\ldots[y_n/x_n]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1) \in \mathcal{R}^{[\![\varrho']\!]}$ and $\varrho' \models \psi_2$. Therefore $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1) \in \mathcal{R}^\varrho$ and $\varrho \models \psi_2$. The case where $\langle Q_1,\ \psi_1\rangle \xrightarrow{\alpha} \langle Q_2,\ \psi_2\rangle$ is similar.

   Now assume that $\langle P_1,\ \phi_1\rangle \xrightarrow{\beta} \langle P_2,\ \phi_2\rangle$, for $\beta ::= c(x),\ x := f(t)$ or $x := new(-)$, and let ϱ_1 be a valuation consistent with $R[x]$ and such that $\varrho_1 \models \phi_2$. Also, assume that $\mathcal{R}^{\varrho_1} = \mathcal{R}^{[\![\varrho_1']\!]}$. Then, by Lemma 4.4, the valuation ϱ_1' is consistent with $R[x]$ and $\varrho_1' \models \phi_2$ (since $\varrho_1 \equiv \varrho_1'$). Therefore, there is a transition $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta'} \langle Q_2,\ \psi_2\rangle$, for some $\beta' = \beta[y_1/x_1]\ldots[y_n/x_n][y/x]$ with $(x_i, y_i) \in R_1$, such that $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(x,y)]) \in \mathcal{R}^{[\![\varrho_1'[x/y]]\!]}$ and $\varrho_1'[x/y] \models \psi_2$. Thus, by Lemma 4.4, we have $(\langle P_2,\ \phi_2\rangle, \langle Q_2,\ \psi_2\rangle, R_1[(x,y)]) \in \mathcal{R}^{\varrho_1[x/y]}$ and $\varrho_1[x/y] \models \psi_2$. The case where $\langle Q_1,\ \psi_1\rangle \xrightarrow{\beta} \langle Q_2,\ \psi_2\rangle$ is similar.

□

Theorem 4.7 allows us to construct bisimulation between constrained processes using only finitely many valuations (one from each equivalence class $[\![\varrho]\!]$). Therefore, it gives us a finite algorithm for verifying bisimilarity between constrained processes.

Example 4.8 *Consider processes P and Q defined as follows:*

$$P ::= c(x_1).P', \quad P' ::= c(x_2).P \quad \textit{and} \quad B ::= c(y).B\ .$$

We show that the constrained processes $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$ are bisimilar. (Symbolic semantics of $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$ are given in Fig. 4.)

$$\langle P,\ \mathbf{1}\rangle \xrightarrow{c(x_1)} \langle P',\ \mathcal{M}(x_1)\rangle \xrightarrow{c(x_2)} \langle P,\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle \underset{c(x_2)}{\overset{\overline{c}(x_1)}{\rightleftarrows}} \langle P',\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle$$

$$\langle Q,\ \mathbf{1}\rangle \xrightarrow{c(y)} \langle Q,\ \mathcal{M}(y)\rangle \circlearrowleft^{c(y)}$$

Figure 4: Symbolic Semantics of $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$.

We use Theorem 4.7 in order to establish a bisimulation between $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$ with respect to the empty relation. First we find the set of formulas occurring in $\mathcal{D}(\langle P,\ \phi\rangle)$ and $\mathcal{D}(\langle Q,\ \psi\rangle)$:

$$\{\mathbf{1}, \mathcal{M}(x_1), \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\} \quad \{\mathbf{1}, \mathcal{M}(y)\},$$

thus, we only need to consider sets of variables $\{x_1, x_2\}$ and $\{y\}$. Since those formulas are satisfied by every valuation, there are only five equivalence classes with respect to $\equiv$:

$$\begin{aligned}
\llbracket\rho_1\rrbracket &= \{\varrho' \mid \varrho'(x_1) \neq \varrho'(x_2),\ \varrho'(x_1) \neq \varrho'(y) \textit{ and } \varrho'(x_2) \neq \varrho'(y)\}\\
\llbracket\rho_2\rrbracket &= \{\varrho' \mid \varrho'(x_1) = \varrho'(x_2),\ \varrho'(x_1) \neq \varrho'(y) \textit{ and } \varrho'(x_2) \neq \varrho'(y)\}\\
\llbracket\rho_3\rrbracket &= \{\varrho' \mid \varrho'(x_1) = \varrho'(y),\ \varrho'(x_1) \neq \varrho'(x_2) \textit{ and } \varrho'(x_2) \neq \varrho'(y)\}\\
\llbracket\rho_4\rrbracket &= \{\varrho' \mid \varrho'(x_2) \neq \varrho'(y),\ \varrho'(x_1) \neq \varrho'(x_2) \textit{ and } \varrho'(x_2) = \varrho'(y)\}\\
\llbracket\rho_5\rrbracket &= \{\varrho' \mid \varrho'(x_1) = \varrho'(x_2) = \varrho'(y)\}.
\end{aligned}$$

For each equivalence class $\llbracket\varrho_i\rrbracket$*, the relation* $\mathcal{R}^{\llbracket\varrho_i\rrbracket}$ *are constructed in Fig. 5. The bisimulation ends since the last triplet has already been visited. Therefore, we obtain the symbolic bisimulation* $\mathcal{R} = \{\mathcal{R}^{\llbracket\rho_i\rrbracket}\}_{i=1,2,3,4,5}$ *where*

$$\begin{aligned}
\mathcal{R}^{\llbracket\rho_1\rrbracket} &= \{\ (\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset)\ \}\\
\mathcal{R}^{\llbracket\rho_2\rrbracket} &= \{\ (\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset)\ \}\\
\mathcal{R}^{\llbracket\rho_3\rrbracket} &= \{\ (\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset),\ (\langle P',\ \mathcal{M}(x_1)\rangle, \langle Q,\ \mathcal{M}(y)\rangle, \{(x_1,y)\})\ \}\\
\mathcal{R}^{\llbracket\rho_4\rrbracket} &= \{\ (\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset)\ \}\\
\mathcal{R}^{\llbracket\rho_5\rrbracket} &= \{\ (\langle P,\ \mathbf{1}\rangle, \langle Q,\ \mathbf{1}\rangle, \emptyset),\ (\langle P',\ \mathcal{M}(x_1)\rangle, \langle Q,\ \mathcal{M}(y)\rangle, \{(x_1,y)\}),\\
&\quad (\langle P,\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle, \langle Q,\ \mathcal{M}(y)\rangle, \{(x_1,y),(x_2,y)\}),\\
&\quad (\langle P',\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle, \langle Q,\ \mathcal{M}(y)\rangle, \{(x_1,y),(x_2,y)\})\ \}.
\end{aligned}$$

$\mathcal{R}^{\llbracket\rho_1\rrbracket}$

$\mathcal{D}(\langle P,\ \mathbf{1}\rangle)$	$\mathcal{D}(\langle Q,\ \mathbf{1}\rangle)$	R
$\langle P,\ \mathbf{1}\rangle$	$\langle Q,\ \mathbf{1}\rangle)$	$\emptyset$

`go to relations` $\mathcal{R}^{\llbracket\rho_3\rrbracket}$ and $\mathcal{R}^{\llbracket\rho_5\rrbracket}$

$\mathcal{R}^{\llbracket\rho_2\rrbracket}$

$\mathcal{D}(\langle P,\ \mathbf{1}\rangle)$	$\mathcal{D}(\langle Q,\ \mathbf{1}\rangle)$	R
$\langle P,\ \mathbf{1}\rangle$	$\langle Q,\ \mathbf{1}\rangle)$	$\emptyset$

`go to relations` $\mathcal{R}^{\llbracket\rho_3\rrbracket}$ and $\mathcal{R}^{\llbracket\rho_5\rrbracket}$

$\mathcal{R}^{\llbracket\rho_3\rrbracket}$

$\mathcal{D}(\langle P,\ \mathbf{1}\rangle)$	$\mathcal{D}(\langle Q,\ \mathbf{1}\rangle)$	R
$\langle P,\ \mathbf{1}\rangle$	$\langle Q,\ \mathbf{1}\rangle)$	$\emptyset$
$\langle P',\ \mathcal{M}(x_1)\rangle$	$\langle Q,\ \mathcal{M}(y))\rangle$	$\{(x_1,y)\}$

`go to relation` $\mathcal{R}^{\llbracket\rho_5\rrbracket}$

$\mathcal{R}^{\llbracket\rho_4\rrbracket}$

$\mathcal{D}(\langle P,\ \mathbf{1}\rangle)$	$\mathcal{D}(\langle Q,\ \mathbf{1}\rangle)$	R
$\langle P,\ \mathbf{1}\rangle$	$\langle Q,\ \mathbf{1}\rangle)$	$\emptyset$

`go to relations` $\mathcal{R}^{\llbracket\rho_3\rrbracket}$ and $\mathcal{R}^{\llbracket\rho_5\rrbracket}$

$\mathcal{R}^{\llbracket\rho_5\rrbracket}$

$\mathcal{D}(\langle P,\ \mathbf{1}\rangle)$	$\mathcal{D}(\langle Q,\ \mathbf{1}\rangle)$	R
$\langle P,\ \mathbf{1}\rangle$	$\langle Q,\ \mathbf{1}\rangle)$	$\emptyset$
$\langle P',\ \mathcal{M}(x_1)\rangle$	$\langle Q,\ \mathcal{M}(y)\rangle$	$\{(x_1,y)\}$
$\langle P,\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle$	$\langle Q,\ \mathcal{M}(y)\rangle$	$\{(x_1,y),(x_2,y)\}$
$\langle P',\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle$	$\langle Q,\ \mathcal{M}(y)\rangle$	$\{(x_1,y),(x_2,y)\}$
$\langle P,\ \mathcal{M}(x_1) \wedge \mathcal{M}(x_2)\rangle$	$\langle Q,\ \mathcal{M}(y)\rangle$	$\{(x_1,y),(x_2,y)\}$

`halt: already visited (success)`

Figure 5: Symbolic Bisimulation of $\langle P,\ \mathbf{1}\rangle$ and $\langle Q,\ \mathbf{1}\rangle$.

5 Future Work and Related Work

This paper presents a symbolic framework for the analysis of security protocols. It is based on a message algebra that handles cryptographic primitives and a logic over this message algebra. The notion of constrained processes is then introduced as a value-passing process paired with a formula. Processes are defined through SPPA, a process algebra which allows for the specification of local function calls as visible actions. SPPA also gives, through markers actions, a clearer view of communication between principals. Generating functions for random numbers, fresh nonces and fresh keys, are introduced into SPPA's syntax in order to specify intruders generating fake addresses and fake messages. From SPPA symbolic semantics for constrained processes, we then establish a bisimulation equivalence. Apart from introducing a new symbolic approach, the two major results of this paper are that every formula obtained in our logic is decidable (Theorem 2.1), and that the bisimulation equivalence between constrained processes corresponds to Milner's strong bisimulation between value-passing processes (Theorem 4.3). Another main result of this paper is a sound and complete method, called symbolic bisimulation, to check for bisimilarity between constrained processes.

The main difference between our approach and Hennessy-Lin's [HL95] is the symbolic transition system: in our symbolic framework, we assign to each state (process) a formula giving a precise description of the free variables involved in the process; Hennessy-Lin's symbolic framework requires considering the formula built from some path leading to a given state (process). Our symbolic framework was developed with security analysis in mind - it is then essential to have an accurate description of the symbolic values at a given state in order to properly analyse a security protocol in a computer system. Indeed, security protocol analysis often requires checking the effect of random values (e.g. nonces, fresh keys or fake messages) on certain principals of the protocol. In this context, our notion of constrained process allows us to explicitly view which such random value could lead, at a certain point of the protocol, to either a confidentiality leak or a masquerade (authentication attack). For instance, in denial of service analysis, we commonly need to verify whether a fake message sent by an intruder can cause the execution of a function requiring a great amount of resource (e.g. decryption or signature checking). In that case, one strategy based on constrained processes would be to verify, for every process following such costly action, the restriction imposed by the formula to the variable representing the fake message: if every fake messages satisfy the formula, then we should conclude that the protocol can not detect fake protocol runs; if only few fake messages satisfy the formula, then we should conclude that the protocol is safe since most fake protocol runs initiated by an intruder will have been detected previously. A similar method, based on SPPA, for detecting denial of service vulnerabilities was introduce in a previous paper [LM03].

Other significant symbolic method applied to security protocols were proposed by Boreale [Bor01] and Fiore-Abadi [FA01]. Starting from a process algebra similar to spi-calculus, Boreale introduces a symbolic operational semantics based on unification. Boreale then give a method carrying out trace analysis directly on the symbolic model. Also starting from a process algebra similar to spi-calculus, Fiore and Abadi propose a decision procedure for knowledge checking and a symbolic procedure for knowledge analysis. In future work, we plan to establish more complete relationships using these methods. It would require introducing constrained processes containing π-caculus and spi-calculs processes, and establishing a symblic semantics for such constrained processes.

Bibliography

[AG98] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, Winter 1998.

[AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of ICALP'01*, 2001.

[BNP99] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Logic in Computer Science*, pages 157–166, 1999.

[Cor02] V. Cortier. Observational equivalence and trace equivalence in an extension of spi-calculus. application to cryptographic protocols analysis. Technical Report LSV-02-3, Lab. Specification and Verification, ENS de Cachan, Cachan, France, March 2002.

[FA01] M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *Proceedings of CSFW'01*, 2001.

[FGC97] R. Focardi, A. Ghelli, and R. Gorrieri. Using non interference for the analysis of security protocols. In H. Orman and C. Meadows, editors, *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, September 1997.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[FM99] R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *Proceedings of World Congress on Formal Methods (FM'99)*, volume 1708 of *LNCS*, pages 794–813. Springer, 1999.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[LM02] S. Lafrance and J. Mullins. Bisimulation-based non-deterministic admissible interference and its application to the analysis of cryptographic protocols. In James Harland, editor, *Electronic Notes in Theoretical Computer Science*, volume 61. Elsevier Science Publishers, 2002.

[LM03] S. Lafrance and J. Mullins. Using admissible interference to detect denial of service vulnerabilities. In *Sixth International Workshop in Formal Methods*. Electronic Workshops in Computing (eWiC) by British Computer Society (BCS), 2003. (*to appear*). Also available at `www.crac.polymtl.ca`.

[Low96] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Proceedings of TACAS'96*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[MPD92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, September 1992.

[Mul00] J. Mullins. Nondeterministic admissible interference. *Journal of Universal Computer Science*, 6(11):1054–1070, 2000.

[Sch96] S. Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

A Operational Semantics of SPPA.

The operational semantics of SPPA, given in Fig. 6, is a value-passing-based semantics defined only for closed processes i.e., process P such that $fv(P) = \emptyset$.

`Input`	$\dfrac{a \in \mathcal{M}}{c(x).P \xrightarrow{c(a)} P[a/x]}$	`Function`	$\dfrac{a'{=}f(a) \quad \text{and} \quad f \in \mathcal{F}_{id_P}}{x{:=}f(a).P \xrightarrow{a'{:=}f(a)} P[a'/x]}$
`Output`	$\dfrac{-}{\overline{c}(a).P \xrightarrow{\overline{c}(a)} P}$	`Protocol`	$\dfrac{P \xrightarrow{\alpha} P' \quad \text{and } \alpha \notin C}{P \Vert Q \xrightarrow{\alpha} P' \Vert Q}$
`Match`	$\dfrac{P \xrightarrow{\alpha} P'}{[a{=}a]P \xrightarrow{\alpha} P'}$	`Synchronisation`	$\dfrac{P \xrightarrow{\overline{c}(a)} P' \quad \text{and} \quad Q \xrightarrow{c(a)} Q'}{P \Vert Q \xrightarrow{\overline{\delta}^c_{id_1}(a)} P' \Vert Q' \quad \xrightarrow{\delta^c_{id_2}(a)} P' \Vert Q'}$
`Sum`	$\dfrac{P \xrightarrow{\alpha} P'}{P{+}Q \xrightarrow{\alpha} P'}$	`Restriction`	$\dfrac{P \xrightarrow{\alpha} P' \quad \text{and} \quad \alpha \notin L}{P \backslash L \xrightarrow{\alpha} P' \backslash L}$
`Parallel`	$\dfrac{P \xrightarrow{\alpha} P'}{P \vert Q \xrightarrow{\alpha} P' \vert Q}$	$\mathcal{O}$`-Observation`	$\dfrac{P \xrightarrow{\gamma} P' \quad \text{and} \quad \gamma \in \mathcal{O}^{-1}(\alpha)}{P/\mathcal{O} \xrightarrow{\alpha} P'/\mathcal{O}}.$

Figure 6: *Semantics of SPPA processes.*

A process P' is a *derivative* of P if there is a computation $P \xrightarrow{\gamma} P'$ for some $\gamma \in Act^*$. We shall frequently make use of the set

$$\mathcal{D}(P) = \{P' \mid \exists_{\gamma \in Act^*}\ P \xrightarrow{\gamma} P'\}$$

the set of P's derivatives.

B Decidability of Formulas

A formula ϕ is decidable whenever we can a finite algorithm allowing to verify, for every valuation ϱ, whether $\varrho \models \phi$. Therefore, in order to prove that every formula from our logic is decidable, it is enough to show that every closed formula ϕ is decidable i.e., to give an algorithm allowing to decide if $\models \phi$. The first step toward proving this result consists of showing that every closed formula is equivalent to a quantifier-free (closed) formula.

Given a formula $\phi ::= (\exists_x\ \phi_1) \wedge \phi_2$ and a variable y that does not occur in ϕ_2, it is easily seen that ϕ is equivalent to the formula $\exists_y\ (\phi_1[y/x] \wedge \phi_2)$. Hence, we may assume that a closed formula ϕ is always given in its normal form, i.e.

$$\phi ::= \ \exists_{x_1} \ldots \exists_{x_n}\ (\phi_1 \wedge \cdots \wedge \phi_m)$$

where the ϕ_i are either predicates ($\phi_i ::= \mathcal{K}(t),\ \mathcal{I}(t),\ \mathcal{N}(t)$ or $\mathcal{M}(t)$) or equations ($\phi_i =\ t == t'$) with $fv(t), fv(t') \subseteq \{x_1, \ldots, x_n\}$. Moreover, from the definition of $\models (a == b)$ given in Section 2.2, we see that every equation $t == t'$ is equivalent to a finite conjunction of irreducible equations $x == t''$. Thus, we may assume that $\phi ::= \exists_{x_1} \ldots \exists_{x_n} (\phi_1 \wedge \cdots \wedge \phi_m)$ where the sub-formulas ϕ_i are either predicates or equations $x == t_i$ (for $x \in \{x_1, \ldots, x_n\}$ and for some term t_i such that $fv(t_i) \subseteq \{x_1, \ldots, x_n\}$). For the following, we consider the family of closed formulas

$$\begin{aligned}\mathcal{F} \ = \ & \{\exists_{x_1} \ldots \exists_{x_n} (\phi_1 \wedge \cdots \wedge \phi_m) \mid \phi_i ::= \mathcal{K}(t),\ \mathcal{I}(t),\ \mathcal{N}(t), \mathcal{M}(t) \text{ or } x == t, \\ & \quad \text{for some } n, m,\ x \in \{x_1, \ldots, x_n\} \text{ and } fv(t) \subseteq \{x_1, \ldots, x_n\}\} \ \cup\ \{\mathbf{0}, \mathbf{1}\}\end{aligned}$$

Therefore, given a closed formula ϕ, we may always assume that $\phi \in \mathcal{F}$; otherwise, an equivalent formula $\phi' \in \mathcal{F}$ can be easily constructed from ϕ following the steps above.

Given a formula $\phi \in \mathcal{F}$, we can construct an equivalent quantifier-free formula as follows. We proceed by induction on the number of existential quantifier in ϕ. First assume that one of the $\phi_i ::= \ x == t$ (let say $i = 1$). If $t = x$ (i.e. $\phi_1 ::= \ x == x$), then we may drop ϕ_1 and assume that $\phi \ ::= \ \exists_x \exists_{x_1} \ldots \exists_{x_n} (\phi_2 \wedge \cdots \wedge \phi_m)$. Otherwise, if x occurs in t, then ϕ is equivalent to $\mathbf{0}$, hence $\not\models \phi$, since we do not allow infinite messages. If x does not occur in t, then we see that ϕ is equivalent to the formula

$$\exists_{x_1} \ldots \exists_{x_n} (\phi_2[t/x] \wedge \cdots \wedge \phi_m[t/x])$$

which has one less quantifier than ϕ. Moreover, since this formula belongs to $\mathcal{F}$, we conclude the proof by applying the induction hypothesis to it.

Now assume that none of the ϕ_i are equation involving x. Moreover, assume that x occurs only in $\phi_1, \ldots, \phi_k$ (for $k \leq n$). Since $\exists_x \mathcal{I}(t)$ and $\exists_x \mathcal{N}(t)$ can only be true whenever $t = x$, and $\exists_x \mathcal{K}(t)$ can only be true whenever $t = h^n(x)$ for some n, none of the other quantified variables $x_1, \ldots, x_m$ occurs in the predicates $\phi_1, \ldots, \phi_k$. The formula ϕ is therefore equivalent to

$$\exists_x (\phi_1 \wedge \cdots \wedge \phi_k) \ \wedge \ \exists_{x_1} \ldots \exists_{x_m} (\phi_{k+1} \wedge \cdots \wedge \phi_n)$$

where $\phi_j \in \{\mathcal{K}(h^n(x)), \mathcal{I}(x), \mathcal{N}(x)\}$ (for $1 \leq j \leq k$). Hence, it is enough to find a quantifier-free formula equivalent to $\exists_x (\phi_1 \wedge \cdots \wedge \phi_k)$. This formula is equivalent to $\mathbf{1}$ whenever

- $\phi_j = \mathcal{I}(x)$, for every $j = 1, \ldots, k$;
- $\phi_j = \mathcal{N}(x)$ or $\phi_j = \mathcal{K}(h^n(x))$, for every $j = 1, \ldots, k$.

Otherwise it is equivalent to $\mathbf{0}$. Finally, it is straightforward to see that the resulting formula still belongs to $\mathcal{F}$.

Lemma B.1 *Every quantifier-free formula from $\mathcal{F}$ is decidable.*

Proof: Let $\phi ::= \ \phi_1(t_1) \wedge \cdots \wedge \phi_n(t_n)$ be a quantifier free formula from $\mathcal{F}$. Let $fv(\phi) = \{x_1, \ldots, x_m\}$ and let $a_1, \ldots, a_m \in \mathcal{M}$. We proceed by induction on n. The case where $n = 0$, i.e. $\phi ::= \mathbf{0}$ or $\mathbf{1}$ is trivial. Thus, let $n \geq 1$. First assume that $\phi_1 ::= \ \mathcal{I}(t_1)$ (resp. $\mathcal{K}(t_1)$ or $\mathcal{N}(t_1)$). Then $\not\models \phi$ whenever $t_1[a_1/x_1, \ldots, a_m/x_m] \notin \mathcal{I}$ (resp. $t_1[a_1/x_1, \ldots, a_m/x_m] \notin \mathcal{K}$ or $t_1[a_1/x_1, \ldots, a_m/x_m] \notin \mathcal{N}$), which we assumed to be decidable. Otherwise ϕ is equivalent to $\phi_2(t_2) \wedge \cdots \wedge \phi_n(t_n)$ (or $\mathbf{1}$ if $n = 1$). Now assume that ϕ_1 is an equation $x_i == t$. Then we saw that the equation $a_i == t'[a_1/x_1, \ldots, a_m/x_m]$ is decidable. Thus, if $\not\models a_i == t'[a_1/x_1, \ldots, a_m/x_m]$, then $\not\models \phi$. Otherwise ϕ is equivalent to $\phi_2(t_2) \wedge \cdots \wedge \phi_n(t_n)$ (or $\mathbf{1}$ in the case $n = 1$). $\square$

The proof of Theorem 2.1 follows Lemma B.1 and the fact that every closed formula is equivalent to a formula from $\mathcal{F}$.

C Proofs of Lemma 3.3, Lemma 3.4 and Lemma 3.5

Proof of Lemma 3.3: In order to shorten the proof, the two statements are proved simultaneously by induction on the structure of P. Depending on the statement proved, we put $Q' \ ::= \ P'[a_1/x_1] \ldots [a_n/x_n]$ or $Q' \ ::= \ P'[a_1/x_1] \ldots [a_n/x_n][a/x]$. The case where $P \ ::= \ \mathbf{0}$ is trivial. If $P \ ::= \ \alpha'.P'$ or $P \ ::= \ \beta'.P'$, then the conclusion follows from semantics rules `Output`, `Input`, `Function` and `Generator`.

If $P ::= \ A_1 + A_2$, $A_1 | A_2$ or $A_1 \parallel \cdots \parallel A_k$ (and α is not a marker action), then, from semantics rules `Sum`, `Parallel` and `Protocol`, we may assume that

$$A_1[a_1/x_1] \ldots [a_n/x_n] \xrightarrow{\alpha} A_1'[a_1/x_1] \ldots [a_n/x_n]$$

$$\text{(resp. } A_1[a_1/x_1] \ldots [a_n/x_n] \xrightarrow{\beta} A_1'[a_1/x_1] \ldots [a_n/x_n][a/x])$$

with $P' ::= A_1' \parallel A_2 \parallel \cdots \parallel A_k$. Thus, by induction hypothesis, $\langle A_1,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle A_1',\ \phi'\rangle$. Therefore, $\langle P,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi'\rangle$. If $P ::= A_1 \parallel A_2 \parallel \cdots \parallel A_k$ and α is a marker action, then we may assume that

$$A_1[a_1/x_1]\dots[a_n/x_n] \overset{\overline{c}(a)}{\longrightarrow} A_1'[a_1/x_1]\dots[a_n/x_n]$$

and

$$A_2[a_1/x_1]\dots[a_n/x_n] \overset{c(a)}{\longrightarrow} A_2'[a_1/x_1]\dots[a_n/x_n][a/x]$$

with $P' ::= A_1' \parallel A_2' \parallel \cdots \parallel A_k$. By induction hypothesis, we have $\langle A_1,\ \phi\rangle \overset{\overline{c}(t)}{\longrightarrow} \langle A_1',\ \phi'\rangle$ (where $a = t[a_1/x_1]\dots[a_n/x_n]$) and $\langle A_2,\ \phi\rangle \overset{c(x)}{\longrightarrow} \langle A_2',\ \phi'\rangle$. Hence $\langle P,\ \phi\rangle \overset{\alpha'}{\longrightarrow} \langle P',\ \phi'\rangle$ by `Synchronisation`.

If $P ::= [t = t']P_1$ (with $t[a_1/x_1]\dots[a_n/x_n] = t'[a_1/x_1]\dots[a_n/x_n]$), then $P_1[a_1/x_1]\dots[a_n/x_n] \overset{\alpha \ / \ \beta'}{\longrightarrow} Q'$ and, by induction hypothesis, we see that $\langle P_1,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi'\rangle$. Therefore, by semantics rules `Match`, $\langle P,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi'\rangle$ with $\phi' \not\Leftrightarrow \mathbf{0}$ since $\varrho \models t == t'$ for any ϱ such that $\varrho(x_i) = a_i$.

If $P ::= P_1 \setminus L$ (with $\alpha, \beta \notin L$), then $P_1[a_1/x_1]\dots[a_n/x_n] \overset{\alpha \ / \ \beta}{\longrightarrow} Q'$ and, by induction hypothesis, we have $\langle P_1,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi''\rangle$ with $\phi'' \not\Leftrightarrow \mathbf{0}$. Therefore, $\langle P,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi'\rangle$ with $\phi' \not\Leftrightarrow \mathbf{0}$ since $\varrho \models \phi^L_{\alpha' \ / \ \beta'}$ for any ϱ such that $\varrho(x_i) = a_i$.

Finally, if $P ::= P_1/\mathcal{O}$, then there is a computation

$$P_1[a_1/x_1]\dots[a_n/x_n] \overset{\gamma}{\longrightarrow} P_1'[a_1/x_1]\dots[a_n/x_n]$$

$$(\text{resp. } P_1[a_1/x_1]\dots[a_n/x_n] \overset{\gamma}{\longrightarrow} P_1'[a_1/x_1]\dots[a_n/x_n][a/x])$$

such that $\gamma \in \mathcal{O}^{-1}(\alpha)$ (resp. $\gamma \in \mathcal{O}^{-1}(\beta)$) and with $Q' ::= P_1'[a_1/x_1]\dots[a_n/x_n]/\mathcal{O}$ (resp. $Q' ::= P_1'[a_1/x_1]\dots[a_n/x_n][a/x]/\mathcal{O}$). Thus, by induction hypothesis, we have $\langle P_1,\ \phi\rangle \overset{\gamma'}{\longrightarrow} \langle P_1',\ \phi'\rangle$ where $\gamma = \gamma'[a_1/x_1]\dots[a_n/x_n]$ (resp. $\gamma = \gamma'[a_1/x_1]\dots[a_n/x_n][a/x]$). Hence, by semantics rule $\mathcal{O}$`-Observation` and since $P ::= P_1/\mathcal{O}$ and $P' ::= P_1'/\mathcal{O}$, we see that $\langle P,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P',\ \phi'\rangle$, which concludes the proof. □

Proof of Lemma 3.4: For simplicity, the two statements are proved simultaneously by induction on the structure of P. The case where $P ::= \mathbf{0}$ is trivial. Let ϱ be a valuation such that $\varrho \models \phi'$ and put $Q ::= P[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$ and, depending on the statement proved, $Q' ::= P'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$ or $Q' ::= P'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n][\varrho(x)/x]$. Also put $\alpha = \varrho(\alpha')$ and $\beta = \varrho(\beta')$.

If $P ::= \alpha'.P'$ (resp. $P ::= \beta'.P'$), then we have $Q ::= \alpha.Q'$ (resp. $Q ::= \beta.Q'$), thus $Q \overset{\alpha \ / \ \beta}{\longrightarrow} Q'$.

If $P ::= P_1 + P_2$, $P ::= P_1|P_2$, or $P ::= P_1 \parallel P_2$, then, respectively, $Q ::= Q_1 + Q_2$, $Q ::= Q_1|Q_2$ or $Q ::= Q_1 \parallel Q_2$ where $Q_1 ::= P_1[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$ and $Q_2 ::= P_2[\varrho(x_1)/x_1]\dots[\varrho(x_1)/x_n]$. Whenever α' is not a marker action, we may assume that $\langle P_1,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle P_1',\ \phi'\rangle$ with, respectively, $P' ::= P_1'$, $P' ::= P_1'|P_2$ or $P' ::= P_1' \parallel P_2$. Hence, by induction hypothesis, we see that

$$Q_1 \overset{\alpha \ / \ \beta}{\longrightarrow} P_1'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n][\varrho(x)/x],$$

thus, $Q \overset{\alpha \ / \ \beta}{\longrightarrow} Q'$. Now assume that α' is marker action, with $P ::= P_1 \parallel P_2$. Then we may assume that $\langle P_1,\ \psi_1\rangle \overset{c(x)}{\longrightarrow} \langle P_1',\ \psi_1'\rangle$ and $\langle P_2,\ \psi_2\rangle \overset{\overline{c}(t)}{\longrightarrow} \langle P_2',\ \psi_2'\rangle$ with $\phi ::= \psi_\wedge \psi_2$. By induction hypothesis, we have

$$Q_1 \overset{c(a)}{\longrightarrow} P_1'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n][a/x]$$

and

$$Q_2 \overset{\overline{c}(a)}{\longrightarrow} P_2'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$$

where $a = \varrho(t)$. Therefore, $P \overset{\alpha}{\longrightarrow} P'$ by `Synchronisation`.

If $P ::= [t = t']P_1$, then we must have $\langle P_1,\ \phi\rangle \overset{alpha'}{\longrightarrow} \langle P',\ \psi\rangle$ with $\varrho \models \psi$ and $\varrho \models t == t'$ since $\varrho \models \phi'$. Therefore, by induction hypothesis, we have $\langle Q_1,\ \phi\rangle \overset{\alpha' \ / \ \beta'}{\longrightarrow} \langle Q',\ \psi\rangle$ where $Q_1 ::= P_1[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$. Thus, by semantics rules `Match`, $Q \overset{\alpha \ / \ \beta}{\longrightarrow} Q'$ since $\varrho(t) = \varrho(t')$.

If $P ::= P_1 \setminus L$, then we must have $\langle P_1,\ \phi\rangle \overset{alpha'}{\longrightarrow} \langle P_1',\ \psi\rangle$ with $P' ::= P_1' \setminus L$ and $\phi ::= \psi \wedge \phi^L_{\alpha'}$ (resp. $\phi ::= \psi \wedge \phi^L_{\alpha'}$). Furthermore, we see that $Q ::= Q_1 \setminus L$ where $Q_1 ::= P_1[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$. Since $\varrho \models \psi$, we see, by induction hypothesis, that

$$Q_1 \overset{\alpha}{\longrightarrow} P_1'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n]$$

$$(resp. Q_1 \overset{\beta}{\longrightarrow} P_1'[\varrho(x_1)/x_1]\dots[\varrho(x_n)/x_n][\varrho(x)/x]).$$

But $\varrho \models \phi'$, therefore $\varrho \models \phi^L_{\alpha'}$ (resp. $\varrho \models \phi^L_{\beta'}$), we have $\alpha, \beta \notin L$. Hence $Q \overset{\alpha \ / \ \beta}{\longrightarrow} Q'$.

If $P ::= P_1/\mathcal{O}$, then there is a computation $\langle P_1,\ \phi\rangle \xrightarrow{\gamma'} \langle P_1',\ \phi'\rangle$ such that $\gamma' \in \mathcal{O}^{-1}(\alpha)$ (resp. $\gamma' \in \mathcal{O}^{-1}(\beta)$) and with $P' ::= P_1'/\mathcal{O}$. By induction hypothesis, we see that $Q_1 \xrightarrow{\gamma} Q_1'$ where $Q_1' ::= P_1'[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n]$ (resp. $Q_1' ::= P_1'[\varrho(x_1)/x_1]\ldots[\varrho(x_n)/x_n][\varrho(x)/x]$) and $\gamma = \varrho(\gamma')$. Thus, $Q \xrightarrow{\alpha\ /\ \beta} Q'$ since $\gamma \in \mathcal{O}^{-1}(\alpha)$ (resp. $\gamma \in \mathcal{O}^{-1}(\beta)$). □

Proof of Lemma 3.5: For simplicity, the two statements are proved simultaneously by induction on the structure of P. If the transition $\langle P,\ \phi\rangle \xrightarrow{\alpha'\ /\ \beta'} \langle P',\ \phi'\rangle$ comes from either `Input`, `Output`, `Function` or `Generator`, then we directly see from the definition of ϕ' that $\models \phi'[a_1/x_1]\ldots[a_n/x_n]$ (resp. $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$) whenever $\models \phi[a_1/x_1]\ldots[a_n/x_n]$.

If $P ::= [t = t']P_1$, then we see, for each of the four possible definition for ϕ', that the statement holds since $t[a_1/x_1]\ldots[a_n/x_n] = t'[a_1/x_1]\ldots[a_n/x_n]$. If $P ::= P_1 + P_2$, $P ::= P_1|P_2$, $P ::= P_1 \parallel P_2$ (and α is not a marker action), then we assume that $\phi ::= \phi_1 \wedge \phi_2$ and $\phi' ::= \phi_1' \wedge \phi_2$, with $\langle P_1,\ \phi_1\rangle \xrightarrow{\alpha'\ /\ \beta'} \langle P_1',\ \phi_1'\rangle$. Therefore, since $\models \phi[a_1/x_1]\ldots[a_n/x_n]$, we have $\models \phi_1[a_1/x_1]\ldots[a_n/x_n]$, thus $\models \phi_1'[a_1/x_1]\ldots[a_n/x_n]$ (reps. $\models \phi_1'[a_1/x_1]\ldots[a_n/x_n][a/x]$) by induction hypothesis. Hence $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$ (reps. $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$) since $\models \phi_2[a_1/x_1]\ldots[a_n/x_n]$.

If $P ::= P_1 \setminus L$, then $\phi' ::= \phi_1 \wedge \phi^L_{\alpha'}$. By induction hypothesis, we see that $\models \phi_2[a_1/x_1]\ldots[a_n/x_n]$ (reps. $\models \phi_1[a_1/x_1]\ldots[a_n/x_n][a/x]$), and since $\alpha \notin L$, $\models \phi^L_{\alpha'}[a_1/x_1]\ldots[a_n/x_n]$ (reps. $\models \phi^L_{\beta'}[a_1/x_1]\ldots[a_n/x_n][a/x]$). Therefore, $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$ (reps. $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$). Finally, assume that $P ::= P_1/\mathcal{O}$. Then $\langle P_1,\ \phi\rangle \xrightarrow{\gamma'} \langle P_1',\ \phi'\rangle$, where $P' ::= P_!'$ and $\gamma' \in \mathcal{O}^{-1}(\alpha')$ (resp. $\gamma' \in \mathcal{O}^{-1}(\beta')$). Thus, by using the induction hypothesis for each sub-action of γ', we see that $\models \phi'[a_1/x_1]\ldots[a_n/x_n]$ (resp. $\models \phi'[a_1/x_1]\ldots[a_n/x_n][a/x]$) whenever $\models \phi[a_1/x_1]\ldots[a_n/x_n]$. □

Index