On the Secure Implementation of Security Protocols [⋆]

Pablo Giambiagi and Mads Dam

Swedish Institute of Computer Science
Box 1263, S-164 49 Kista, Sweden
{pablo,mfd}@sics.se

Abstract. We consider the problem of implementing a security protocol in such a manner that secrecy of sensitive data is not jeopardized. Implementation is assumed to take place in the context of an API that provides standard cryptography and communication services. Given a dependency specification, stating how API methods can produce and consume secret information, we propose an information flow property based on the idea of invariance under perturbation, relating observable changes in output to corresponding changes in input. Besides the information flow condition itself, the main contributions of the paper are results relating the admissibility property to a direct flow property in the special case of programs which branch on secrets only in cases permitted by the dependency rules. These results are used to derive an unwinding-like theorem, reducing a behavioral correctness check (strong bisimulation) to an invariant.

1 Introduction

We consider the problem of securely implementing a security protocol given an API providing standard services for cryptography, communication, key- and buffer management. In particular we are interested in the problem of confidentiality, that is, to show that a given protocol implementation which uses standard features for encryption, random number generation, input-output etc. does not leak confidential information provided to it, either because of malicious intent, or because of bugs.

Both problems are real. Malicious implementations (Trojans) can leak intercepted information using anything from simple direct transmission to, e.g., subliminal channels, power, or timing channels. Bugs can arise because of field values that are wrongly constructed, mistaken representations, nonces that are reused or generated in predictable ways, or misused random number generators, to give just a few examples.

Our work starts from the assumption that the protocol and the API is known. The task, then, is to ensure that confidential data is used at the correct times and in the correct way by API methods. The constraints must necessarily be quite rigid and detailed. For instance, a non-constant time API method which is made freely available to be applied to data containing secrets can immediately be used in conjunction with otherwise legitimate output to create a timing leak.

[⋆] This material is based upon work partially supported by the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, under Contract No. F61775-01-C0006, and by the European IST project VerifiCard.

Our approach is to formulate a set of rules, which determine the required dependencies between those API method calls that produce and/or consume secrets. An example of such a dependency rule might be

$$\mathbf{send}(v, \mathrm{outchan}) \leftarrow k := \mathbf{key}(\mathrm{Bob}) \wedge m := \mathbf{receive}(\mathrm{inchan}) \wedge v := \mathrm{enc}(m, k)$$

indicating that, if upon its last invocation of the $\mathbf{receive}$ method with argument inchan the protocol received m (and analogously for $\mathbf{key}$, Bob, and k), then the next invocation of $\mathbf{send}$ with second parameter $\mathrm{outchan}$ must, as its first parameter, receive the encryption of m with key k.

A dependency specification determines an information flow property. The rules determine a required dependency relation between API calls. Assurance, then, must be given that no other flows involving secrets exist. Our approach to this is based on the notion of admissibility, introduced first in [4]. The idea is to extract from the dependency specification a set of system perturbation functions g which will allow a system s processing a secret v to act as if it is actually processing another value of that secret, v'. Then, confidentiality is tantamount to showing that system behaviour is invariant under perturbation, i.e. that

$$s[g] \sim s,$$

where $[g]$ is the system perturbation operator. One problem is that, provided this is licensed by the dependency rules, secrets actually become visible at the external interface. For this reason, the perturbation operator $[g]$ must be able to identify the appropriate cases where this applies, so that internal changes in the choice of secret can be undone.

The paper has two main contributions. First, we show how the idea can be realized in the context of a simple sequential imperative language, IMP0. Secondly we establish results which provide efficient (thought not yet fully automated) verification techniques, and give credence to the claim that admissibility is a good formalisation of confidentiality in this context. In particular, we show that, for the special case of programs which branch on secrets only in cases permitted by the dependency rules, admissibility can be reduced to a direct flow property (an invariant) which we call flow compatibility. Vice versa, we show that under some additional assumptions, flow compatibility can be reduced to admissibility.

This work clearly has strong links to previous work in the area of information flow theory and language-based security (cf. [8]). The idea of invariance under perturbation and logical relations underpins most work on secrecy and information flow theory, though not always very explicitly (cf. [3, 5, 11, 9]). The main point, in contrast e.g. to work by Volpano [10] is that we make no attempt to address information flow of a cryptographic program in absolute terms, but are satisfied with controlling the use of cryptographic primitives according to some external protocol specification. This is obviously a much weaker analysis, but at the same time it reflects well, we believe, the situation faced by the practical protocol implementor.

The rest of the paper is structured as follows. In Section 2, we present IMP0 and introduce the main example used in the paper, a rudimentary credit card payment protocol. In Section 3 we introduce an annotated semantics, used in Section 4 to formalize

Table 1. IMP0: Syntax

Basic values (BVal)	b	$::=$	$n \mid a \mid \mathbf{true} \mid (b_1, \ldots, b_n)$
Values (Val)	v	$::=$	$b \mid \mathbf{xcpt}$
Functions (Fun)	f	$::=$	$\mathrm{pf} \mid \mathrm{h}$
Expressions (Expr)	e	$::=$	$v \mid x \mid (e_1, \ldots, e_n) \mid f\ e$
Commands (Com)	c	$::=$	$\mathbf{skip} \mid \mathbf{throw} \mid x := e \mid c_0; c_1 \mid \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid$ $\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} \mid \mathbf{try}\ c_0\ \mathbf{catch}\ c_1$

the dependency rules. The notion of flow compatibility is presented in Section 5 to describe the direct information flow required by a protocol specification. In Section 6 the main information flow condition, admissibility, is introduced. In Section 7 we state and prove the unwinding theorem, while in Section 8 we further investigate the relation between flow compatibility and admissibility. Finally Section 9 concludes with discussion and related work.

2 A Sequential Imperative Language

In this section we introduce IMP0, the language we use for protocol implementation. The intention is to formalise the basic functionality of simple protocol implementations in as uncontroversial a manner as possible.

Table 1 defines the syntax of IMP0, with variables $x \in \mathrm{Var}$, including the anonymous variable $\_$, primitive function and procedure calls, and primitive data types including natural numbers ($n \in \mathrm{Nat}$) and channels ($a \in \mathrm{Chan}$). The set of primitive function symbols, ranged over by pf, includes the standard arithmetic and logical operators. Each primitive function is assumed to execute in constant time, regardless of its arguments. There are also non-primitive (or API) functions, ranged over by h, for encryption ($\mathbf{enc}$), decryption ($\mathbf{dec}$), extracting a key from a keystore ($\mathbf{key}$), and receiving resp. sending a value on a channel ($\mathbf{receive}$ and $\mathbf{send}$). To each (primitive or non-primitive) function symbol f is associated a binary relation $f \subseteq \mathrm{Val} \times \mathrm{Val}_\perp$ so that $\forall v \in \mathrm{Val}.\ \exists v' \in \mathrm{Val}_\perp.\ f(v, v')$ (i.e. functions may be non-deterministic, and may not terminate), and $f(\mathbf{xcpt}, v)$ iff $v = \mathbf{xcpt}$ (i.e. function invocations propagate exceptions from arguments to results). Moreover, primitive functions are assumed not to have local side effects. Communication effects are brought out using transition labels in the next section.

As a running example we use a greatly simplified version of the 1-Key-Protocol (1KP), a protocol for electronic payments [2]. This example is chosen because it is paradigmatic for many simple e-commerce applets which input a collection of data, some sensitive, some not, performs some cryptographic operations on the data, and then transmits the result on a public channel. In the full version of the paper [6] we use a simple declassifier as a second example.

```
Prog 1 :
  while true do
    ACQ := receive acq; ORDER := receive order; ACC := receive acc;
    try
      PKA := key ACQ;
      _ := send((ACQ, ORDER, enc((ORDER, ACC), PKA)), lookup(merchant))
    catch
      _ := send(“error report”, local)
  end
```

Fig. 1. Payment protocol – sample Customer implementation

2.1 A Simple Payment Protocol

The protocol involves three players: A Customer, a Merchant and an Acquirer (ACQ). The Customer possesses a credit card account (ACC) with which it places an order to the Merchant. The Acquirer is a front-end to the existing credit card clearing/authorization network, that receives payments records from merchants and responds by either accepting or rejecting the request. The Customer is required to encrypt the order and account information with the Acquirer’s public key before sending them to the Merchant.

Figure 1 shows what a simple implementation of the Customer’s side of the payment protocol might look like in IMP0. In general, an implementation needs to deal with a lot more issues than what are explicitly addressed at the protocol specification level. These include: Initialisation and use of cryptographic services, where and how data is stored and addressed, communication services, and error handling. Further, in some applications the protocol implementation may well be bundled with the user interface, in which case a further set of issues arise.

It may be instructive to also show some of the means available to implementations wishing to violate confidentiality. For instance, a hostile implementation might embed account information in the ordering field by replacing line 5 of Figure 1 by

$$\_ := \mathbf{send}(((\mathrm{ACQ}, \mathbf{embed}(\mathrm{ORDER}, \mathrm{ACC}), \mathbf{enc}(\ldots)), \ldots), \ldots)\ ,$$

or it might try to replace good nonces or keys by bad ones, for instance by replacing the same line as before by

$$\_ := \mathbf{send}(((\ldots, \mathbf{enc}((\mathrm{ORDER}, \mathrm{ACC}), \mathrm{PK}_{\mathrm{MERCHANT}})), \ldots), \ldots)\ .$$

There are many other simple ways of building covert channels, such as timing channels, for instance by introducing data-dependent delays, either explicitly, or by exploiting timing properties of library functions.

3 Annotated Semantics

The first challenge is to identify the direct flows and computations on critical data (typically: secrets, keys, nonces, or time stamps). Once this is accomplished, other tech-

Table 2. Annotated semantics, expressions

$$\sigma \vdash x \xrightarrow{\tau} \sigma(x) \qquad \frac{\sigma \vdash \epsilon \xrightarrow{\alpha} \epsilon'}{\sigma \vdash (\ldots,\epsilon,\ldots) \xrightarrow{\alpha} (\ldots,\epsilon',\ldots)} \qquad \frac{[\![w]\!] = \mathbf{xcpt}}{\sigma \vdash (\ldots,w,\ldots) \xrightarrow{\tau} w}$$

$$\frac{\sigma \vdash \epsilon \xrightarrow{\alpha} \epsilon'}{\sigma \vdash f\,\epsilon \xrightarrow{\alpha} f\,\epsilon'} \qquad \frac{\mathrm{pf}([\![w]\!],v)}{\sigma \vdash \mathrm{pf}\ w \xrightarrow{\tau} v : \mathrm{pf}\ w} \qquad \frac{h([\![w]\!],v)}{\sigma \vdash h\,w \xrightarrow{v := h\,w} v : h\,w}$$

niques based on non-interference are brought to bear to handle the indirect flows. The direct flows are tracked using annotations. In particular, we need to identify:

1. The operations that cause critical values to enter the system (such as execution of $\mathbf{receive}\ a$ for some given value of a).
2. The operations that are applied to secrets, once they have been input.

To account for this we provide IMP0 with an annotated semantics. Annotations are intended to reveal how a value has been computed, from its point of entry into the system. For instance, the annotated value

$$347 : \mathbf{enc}(717 : \mathbf{receive}\ a,\ 101 : \mathbf{key}\ 533)$$

is intended to indicate that the value 347 was computed by applying the primitive function $\mathbf{enc}$ to the pair $(717, 101)$ for which the left hand component was computed by evaluating $\mathbf{receive}\ a$, and so on.

Annotated expressions and values are obtained by changing the definition of expressions (resp. values) in Table 1:

Annotated basic values (aBVal)	β	$::=$	$b \mid (\beta_1,\ldots,\beta_n) \mid b : \varphi$
Annotated values (aVal)	w	$::=$	$\beta \mid \mathbf{xcpt} \mid \mathbf{xcpt} : \varphi$
Annotated expressions (aExp)	ϵ	$::=$	$w \mid x \mid (\epsilon_1,\ldots,\epsilon_n) \mid f\epsilon$
Annotations (Ann)	φ	$::=$	fw

Annotations are erased using the operation $[\![w]\!]$ which removes annotations in the obvious way.

Table 2 defines the small-step semantics for expression evaluation. The transition relation has the shape

$$\sigma \vdash \epsilon \xrightarrow{\alpha} \epsilon'\,,$$

where α is an action of the form τ (internal computation step) or $v := fw$ (f is applied to the annotated value w resulting in the value v), and σ is an annotated store, a partial function $\sigma \in \mathrm{aStore} \overset{\Delta}{=} [\mathrm{Var} \to \mathrm{aBVal}]$.

Annotations give only static information in the style "the value v' was computed by evaluating $\mathbf{key}\ v\ :\ \mathbf{receive}\ \mathrm{acq}$", but not information concerning which actual invocations of the $\mathbf{key}$ and $\mathbf{receive}$ functions were involved. However, this information is vital to the subsequent information flow analysis, and so we introduce a notion of context to record the last value returned by some given annotated function call (i.e. annotation).

Definition 1 (Context). A *context* is a partial function $s : [\mathrm{Ann} \to \mathrm{Val}]$.

So, if s is a context then $s\ \varphi$ is the last value returned by the annotated function call φ. Contexts form part of program configurations in the annotated semantics:

Definition 2 (Annotated Configuration).
An *annotated configuration* is a triple $\langle c,\ \sigma,\ s\rangle$ where c is a command, $\sigma \in \mathrm{aStore}$ and $s \in \mathrm{Context}$.

The annotated command-level semantics, which derives transitions of the shape $\langle c,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c',\ \sigma',\ s'\rangle$, is standard in its treatment of commands and stores. Concerning contexts, if α is the action $v := \varphi$, then s' is defined as $s[v/\varphi]$. Details are given in the full version of the paper [6].

4 Dependency Rules

Our approach to confidentiality is to ensure that the direct flows of information follow the protocol specification, and then use information flow analysis to protect against indirect flows. In this section we introduce dependency rules to formalize the permitted, direct flows.

Definition 3 (Dependency Specification). A *dependency specification* is a pair $P = \langle \mathcal{S}, \mathcal{A}\rangle$ where $\mathcal{S} \subseteq \mathrm{Ann}$ is a set of annotations, and $\mathcal{A}$ is a finite set of clauses of the form

$$f\ e \leftarrow x_1 := f_1\ e_1 \wedge \ldots x_n := f_n\ e_n \wedge \psi \qquad (1)$$

where none of the expressions e, $e_1, \ldots, e_n$ mention functions or exceptions, ψ is a boolean expression, and variables in e_i do not belong to $\{x_i, \ldots, x_n\}$.

The intention is that $\mathcal{S}$ represents a set of secret entry points (such as: $\mathbf{receive}\ \mathrm{acc}$), and that the rules in $\mathcal{A}$ represent the required data flow through the program.

A clause in the policy declares a function invocation $f\ e$ to be admissible if the conditions to the right of the arrow are satisfied. Conjuncts of the form $x_i := f_i\ e_i$ are satisfied if variable x_i matches the last input from annotation $f_i\ e_i$. The boolean expression ψ represents an extra condition that relates the values returned by the different function invocations. More precisely, let a context s be given. A *valid substitution* for clause (1) is an annotated store σ such that

1. $\sigma(x_i) = s(f_i\ (e_i\sigma))$: $f_i\ (e_i\sigma)$ for all $i : 1 \leq i \leq n$,
2. for $x \neq x_i$ ($\forall i$: $1 \leq i \leq n$), $\sigma(x)$ has not annotation in $\mathcal{S}$,
3. $\mathrm{eval}(\psi\sigma) = \mathrm{true}$.

That is, boolean conditions are true, and the value bound to x_i is the last value returned by the annotated function call $f_i\ (e_i\sigma)$. By $e\sigma$ we mean the annotated expression (aExpr) that results from substituting $\sigma(x)$ for every variable x in e. Notice that the restrictions on e_i in Def. 3 guarantee that $e_i\sigma$ is an annotated value. The function eval just evaluates the annotated boolean expression $\psi\sigma$ in the expected way.

We can now determine whether a particular function invocation is admitted by the dependency specification.

Definition 4 (Admissible Invocation). Let α be an annotated action of the form $v := f\ w$. A dependency specification $P = \langle \mathcal{S}, \mathcal{A} \rangle$ *admits annotated action* α in context s iff either

1. no annotation in w belongs to $\mathcal{S}$ (that is, the output does not depend directly on any secret annotation), or
2. there is a clause $f\ e \leftarrow x_1 := f_1\ e_1 \wedge \ldots \wedge\ x_n := f_n\ e_n\ \wedge\ b$ in $\mathcal{A}$ and a valid substitution σ for this clause such that $e\sigma = w$.

If one of these conditions holds we write $P, s \vdash \alpha$ ok.

Observe that the concept of admissible action covers both those actions whose execution is required by the protocol specification, as well as those that do not (explicitly) involve any sensitive data. In particular, internal τ transitions are always admissible (i.e. $P, s \vdash \tau$ ok).

Example 1 (Dependency Specification for 1KP Clients).
In the simplified version of the 1KP protocol, the only piece of local information that the Customer should protect is her account number. Therefore, $\mathcal{S} = \{\mathbf{receive}\ \mathrm{acc}\}$. Neither the key (which is public), the acquirer's name, nor the order need to be protected. The set $\mathcal{A}$ contains the clauses:

$$\begin{aligned} \mathbf{enc}((y,z),k) &\leftarrow x := \mathbf{receive}\ \mathrm{acq}\ \wedge\ z := \mathbf{receive}\ \mathrm{acc}\ \wedge\ k := \mathbf{key}\ x \\ \mathbf{send}(u,s) &\leftarrow u := \mathbf{enc}((y,z),k) \end{aligned}$$

The first clause expresses when an invocation of the encryption function is admissible. In this example, encryption is used just once in each protocol run, but in general this might not be so. Moreover, since invocation of the encryption function, as any other function with a non-constant execution time, could be used to create a timing leak, the dependency specification does need to say under which circumstances it may be invoked, apart from its usage in the main input-output flow.

Notice how the variables y and s are not bound to the right of the clauses, reflecting the fact that we do not put any requirement on the format of the order and neither its destination (since it is intended for transmission in the clear anyway), beyond the restriction that it should not be used to encode secret information.

Let now $P = \langle \mathcal{S}, \mathcal{A} \rangle$.

– Let $\alpha_1 = b_1 := \mathbf{receive}\ \mathrm{acq}$. Then $P, s \vdash \alpha_1$ ok for any s since no annotation in acq belongs to $\mathcal{S}$.

– Let $\alpha_4 = b_4 := \mathbf{enc}((b, b_2 : \mathbf{receive}\ \mathrm{acc}), b_3 : \mathbf{key}\ (b_1 : \mathbf{receive}\ \mathrm{acq}))$. Consider a context s where $s(\mathbf{receive}\ \mathrm{acq}) = b_1$, $s(\mathbf{receive}\ \mathrm{acc}) = b_2$ and $s(\mathbf{key}\ b_1 : \mathbf{receive}\ \mathrm{acq}) = b_3$. Then $P, s \vdash \alpha_4$ ok since we find the substitution σ mapping x to b_1, z to b_2, k to b_3 and y to b, validating the condition 4.2. If on the other hand e.g. $s(\mathbf{receive}\ \mathrm{acq}) = b_5 \neq b_1$ then the condition would be violated and α_4 would not be admissible in the context s.

As the example show, dependency specifications are very low-level objects. They are not really meant as external specifications of confidentiality requirements, but rather as intermediate representations of flow requirements, generated from some more user-friendly protocol specification once a specific runtime platform has been chosen.

5 Flow Compatibility

Dependency specifications determine, through Definition 4, when a function invocation is admissible. In this section we tie this to the transition semantics to obtain an account of the direct information flow required by a dependency specification.

Let the relation

$$\langle c, \sigma, s\rangle \Rightarrow \langle c', \sigma', s'\rangle$$

be the reflexive, transitive closure of the annotated transition relation, i.e. the smallest relation such that $\langle c, \sigma, s\rangle \Rightarrow \langle c', \sigma', s'\rangle$ holds iff either $c = c'$, $\sigma = \sigma'$ and $s = s'$ or else c_1, σ_1, s_1 exists such that $\langle c, \sigma, s\rangle \Rightarrow \langle c_1, \sigma_1, s_1\rangle$ and $\langle c_1, \sigma_1, s_1\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle$.

Definition 5. Let the dependency specification $P = \langle \mathcal{S}, \mathcal{A}\rangle$ be given. The command $c \in$ *Com is flow compatible with* P for initial store σ and initial context s, if whenever $\langle c,\ \sigma,\ s\rangle \Rightarrow \langle c_1,\ \sigma_1,\ s_1\rangle \xrightarrow{\alpha} \langle c_2,\ \sigma_2,\ s_2\rangle$ then $P, s_1 \vdash \alpha$ ok.

Example 2 (Flow Compatibility for 1KP Client). The command **Prog 1** of Figure 1 is flow compatible with the 1KP client dependency specification of Example 1 above, for any initial store σ. This is seen by proving an invariant showing that whenever execution of **Prog 1** reaches one of the send statements of **Prog 1** then for suitable choices of v_1, v_2 and v_3,

$$s(\mathbf{receive}\ \mathrm{acq}) = v_1 = \sigma(\mathrm{ACQ})$$
$$s(\mathbf{receive}\ \mathrm{acc}) = v_2 = \sigma((ACC))$$
$$s(\mathbf{key}\ x) = v_3 = \sigma(\mathrm{PKA})$$

If we attempt to use a subliminal channel by replacing line 5 (the first send statement) of **Prog 1** by a command such as

$$\_ := \mathbf{send}((\mathrm{ACQ}, \mathbf{embed}(\mathrm{ACC}, \mathrm{ORDER}), \mathbf{enc}((\mathrm{ORDER}, \mathrm{ACC}), \mathrm{PKA})), \mathbf{lookup}(\mathrm{MERCHANT}))\,,$$

then flow compatibility is violated, as expected. On the other hand, the command obtained by adding after the first send statement of **Prog 1** the command

$$\mathbf{if}\ \mathrm{ACC} = \text{“some fixed value } v\text{”}\ \mathbf{then}\ \mathbf{send}(\text{“FOUND!”}, \mathrm{leak\_channel})\ \mathbf{else}\ \mathbf{skip}$$

is flow compatible, also as expected, since the indirect leak will not be traced by the annotation regime.

6 Admissibility

If there is an admissible flow of information from some input, say $\mathbf{receive}\ \mathrm{acc}$, to some output, say, $\mathbf{send}(\ldots, \mathbf{enc}((\ldots, \mathrm{acc}), \ldots), \ldots)$ then by perturbing the input, corresponding perturbations of the output should result, and only those. In this section we formalize this idea.

In the context of multilevel security it is by now quite well understood how to model absence of information flow (from Hi to Lo) as invariance of system behaviour under perturbation of secret inputs (c.f. [3, 5, 11, 9], see also [1] for application of similar ideas in the context of protocol analysis). For instance, the intuition supporting Gorrieri and Focardi's Generalized Noninterference model is that there should be no observable difference (i.e. behaviour should be invariant) whether high-level inputs are blocked or allowed to proceed silently. So the perturbation of high-level inputs, in this case, is whether or not they take place at all.

Here the situation is somewhat different since the multilevel security model is not directly applicable: There is no meaningful way to define security levels reflecting the intended confidentiality policy, not even in the presence of a trusted downgrader. On the contrary, the task is to characterize the admissible flows from high to low in such a manner that no trust in the downgrader (i.e. the protocol implementation) will be required.

The idea is to map a dependency specification to a set of system perturbations. Each such function is a permutation on actions and configurations which will make a configuration containing a secret, say x, appear to the external world as if it actually contains another secret, say x'. If the behaviour of the original and the permuted configuration are the same, the external world will have no way of telling whether the secret is x or x'.

At the core of any configuration permutation there is a function permuting values (e.g. x and x'). This leads to the following definition:

Definition 6 (Value Permutation). A bijection $g\colon \mathrm{aVal} \to \mathrm{aVal}$ is a *value permutation* if it preserves the structure of annotated values:

1. $g(v) = v\,,$
2. $g(\beta_1, \ldots, \beta_n) = (g(\beta_1), \ldots, g(\beta_n))$, and
3. $g(v : f\ w) = v' : f\ g(w)$, for some suitable value v';

and it preserves the meaning of functions:

4. Suppose $g(v : f\ w) = v' : f\ w'$ and that there is at least a value u' s.t. $f([\![w']\!], u')$. Then $f([\![w']\!], v')$, whenever $f([\![w]\!], v)$ or $\nexists u \in \mathrm{Val}.\ f([\![w]\!], u)$.

We extend value permutations over transition labels and contexts. In the first case, let $g(\tau) \triangleq \tau$ and $g(v := \varphi) \triangleq v' := \varphi'$, where $g(v : \varphi) = v' : \varphi'$. For contexts, define

$$g(s)(f\ w) \triangleq [\![g(v' : f\ g(w))]\!], \text{ where } v' = s(f\ g(w))\,.$$

The following lemma establishes the coherence of the above definitions. It states that the relation between contexts s and $g(s)$ is preserved after the execution of action $v := \varphi$, resp. $g(v := \varphi)$.

Lemma 1. *If* $g(v : \varphi) = v' : \varphi'$ *then* $g(s[v/\varphi]) = g(s)[v'/\varphi']$.

Not all value permutations are interesting for our purposes. In fact, we are only interested in those that permute secrets as dictated by a dependency specification.

Definition 7 (Secret Permuter). Assume given a dependency specification P. A *secret permuter for* P is a value permutation g satisfying the following conditions:

1. if $f\ w$ does not contain annotations in $\mathcal{S}$ then $g(v : f\ w) = v : f\ w\,$,
2. if $f\ w \in \mathcal{S}$ then $f\ g(w) \in \mathcal{S}\,$,
3. if $P, s \vdash \alpha\ \mathsf{ok}$ then $P, g(s) \vdash g(\alpha)\ \mathsf{ok}\,$,
4. if $\exists s.\ P, s \vdash v := f\ w$ ok, then
   – $g(\mathbf{xcpt} : f\ w) = \mathbf{xcpt} : f\ g(w)$, and
   – $f[\![w]\!] \Uparrow$ iff $f[\![g\,w]\!] \Uparrow$, where $f\ v \Uparrow$ iff $\nexists v' \in \mathrm{Val}. f(v, v')$
5. $g = g^{-1}$.

As expected, a secret permuter affects only secret values. This is implied by the first condition in Definition 7. According to the second condition, permutations must also stay within the bounds imposed by set $\mathcal{S}$. Condition (7.3) implies that a secret permuter must respect the admissibility predicate so that actions α that are admissible in a context s will remain admissible once both the action and the context have been permuted. On the other hand, if a dependency specification admits a certain function call $f\ w$ (admissible invocation), then we assume that it also permits the observation of f's exceptional and terminating behaviour. Thus, if the execution of $f\ w$ raises an exception (resp. does not terminate), we should not consider those cases where $f\ g(w)$ does not raise an exception (resp. does terminate). This is reflected by condition (7.4).

Finally we impose the requirement that g be a period 2 permutation (7.5). This seems natural given the intuition that the role of g is to interchange values of secrets. Not only does this requirement help simplify several results, but we conjecture that its introduction in Def. 7 represents no loss of generality.

The following lemma and proposition further characterize the set of secret permuters associated to a dependency specification.

Lemma 2. *Let* g *be a secret permuter. Then*

1. $g(g(\alpha)) = \alpha$, *and*
2. $g(g(s)) = s\,$.

Proposition 1 (Composition of Secret Permuters). *Given a dependency specification, the set of secret permuters is closed under functional composition.*

Example 3 (Secret Permuter for the 1KP Example). Let g exchange values as follows:

$$212 : \mathbf{receive}\ \mathrm{acc} \leftrightarrow 417 : \mathbf{receive}\ \mathrm{acc}$$

$$\{b, 212\}_{b_3} : \mathbf{enc}((b, 212 : \mathbf{receive}\ \mathrm{acc}), b_3 : \mathbf{key}\ b_1 : \mathbf{receive}\ \mathrm{acq}) \leftrightarrow$$
$$\{b, 417\}_{b_3} : \mathbf{enc}((b, 417 : \mathbf{receive}\ \mathrm{acc}), b_3 : \mathbf{key}\ b_1 : \mathbf{receive}\ \mathrm{acq})$$

where $\{b\}_{b'}$ represents a value $v \in \mathrm{Val}$ such that $\mathbf{enc}((b, b'),\ v)$. On all other values, g acts in accordance with conditions in Defs. 6 and 7. Conditions (6.1)–(6.4) and (7.x, with $x \neq 3$) are easily validated. To verify condition (7.3) consider the action

$$\alpha = \{b, 212\}_{b_3} := \mathbf{enc}((\ldots, 212 : \mathbf{receive}\ \mathrm{acc}), \ldots)\,.$$

If $P, s \vdash \alpha$ ok then $s(\mathbf{receive}\ \mathrm{acc}) = 212$, by Def. 4. To see that $P, g(s) \vdash g(\alpha)$ ok observe that

$$g(\alpha) = \{b, 417\}_{b_3} := \mathbf{enc}((\ldots, 417 : \mathbf{receive}\ \mathrm{acc}), \ldots)$$

and $g(s)(\mathbf{receive}\ \mathrm{acc}) = 417$ by the definition of $g(s)$, so we can indeed conclude that $P, g(s) \vdash g(\alpha)$ ok.

We have extended secret permuters over transition labels and contexts. Stores and commands can equally be permuted. The extension of a secret permuter g over a store is given by the equation $g(\sigma)(x) = g(\sigma(x))$. For a command c, define $g(c)$ to preserve the structure of the command, down to the level of single annotated values which are permuted according to g. For example, $g(\_ \mathrel{\mathbf{:=}} \mathbf{enc}((b, b_2 : \mathbf{receive}\ \mathrm{acc}), PKA)) = \_ \mathrel{\mathbf{:=}} \mathbf{enc}((b, g(b_2 : \mathbf{receive}\ \mathrm{acc})), PKA)$. Commands like these occur naturally during the course of expression evaluation, which is governed by a small-step semantics.

The idea now is to compare the behaviour of a given command on a given store and context with its behaviour where secrets are permuted internally and then restored to their original values at the external interface, i.e. at the level of actions. For this purpose we introduce a new construct at the command level, perturbation $c[g]$, somewhat reminiscent of the CCS relabelling operator, with the following transition semantics

$$\frac{\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle}{\langle c[g],\ \sigma,\ s\rangle \xrightarrow{[\![g(s,\alpha)]\!]} \langle c'[g], \sigma', s'\rangle} \tag{2}$$

where $[\![v := f\ w]\!] = v := f\ [\![w]\!]$, and $g(s, \alpha)$ permutes α only if it is an admissible invocation (i.e. $g(s, \alpha) = g(\alpha)$, if $P, s \vdash \alpha$ ok; and $g(s, \alpha) = \alpha$, otherwise). So a perturbed command is executed by applying the secret permuter at the external interface, and forgetting annotations. The latter point is important since the annotations describe data flow properties internal to the command at hand; the externally observable behaviour should depend only on the functions invoked at the interface, and the values provided to these functions as arguments.

Notice the use of $g(s, \alpha)$ in (2). The effect of this condition is that actions are only affected by the permuter when they are “ok”. Secret input actions are generally always “ok”, and so in general cause the internal choice of secret to be permuted. Output actions that are not “ok”, however, are not affected by $g(s, \alpha)$, and so in this case a mismatch between value input and output may arise.

Thus, if the behaviour of a command is supposed to be invariant under perturbation, the effect is that it must appear to the external world to behave the same whether or not a secret permuter is applied to the internal values. This is reflected in the following definition.

Definition 8 (Admissibility). A command $c \in \mathit{Com}$ is admissible for the store σ and context s, the dependency specification P, if for all secret permuters g for P:

$$\langle c[I], \sigma, s\rangle \sim \langle g(c)[g], g(\sigma), g(s)\rangle \tag{3}$$

where I is the identity secret permuter and $\sim$ is the standard Park-Milner strong bisimulation equivalence.

Observe that the effect of perturbing a command with the identity secret permuter is just to erase annotations at the interface, but keeping all values intact.

7 Local Verification Conditions

Applying the definition of admissibility out of the box can be quite cumbersome, since it is tantamount to searching for, and checking, a bisimulation relation. In case the control flow is not affected by the choice of secrets one may hope to be able to do better, since only data-related properties need to be checked. In this section we give such a local condition.

Definition 9 (Stability for Commands). Let a dependency specification P be given. Let $\preccurlyeq$ be the smallest reflexive and transitive relation over commands such that, for all commands c_0 and c_1, $c_0 \preccurlyeq c_0; c_1$ and $c_0 \preccurlyeq \mathbf{try}\ c_0\ \mathbf{catch}\ c_1$. The command $c \in \mathit{Com}$ is *stable* if for all $c' \preccurlyeq c$ and for all secret permuter g,

1. if $c' = \mathbf{if}\ \beta\ \mathbf{then}\ c_2\ \mathbf{else}\ c_3$, then $[\![\beta]\!] = [\![g(\beta)]\!]$,
2. if $c' = r[\epsilon]$ and w is a subterm of ϵ, then $[\![w]\!] = \mathbf{xcpt}$ iff $[\![g(w)]\!] = \mathbf{xcpt}$, and
3. if $c' = r[\epsilon]$ and $f\ w$ is a subterm of ϵ, then $f\ [\![w]\!] \Uparrow$ iff $f\ [\![g\ w]\!] \Uparrow$,

where $r[\cdot] ::= x := \cdot \mid \mathbf{if} \cdot \mathbf{then}\ c_0\ \mathbf{else}\ c_1$.

For stable commands we obtain strong properties concerning the way secret permuters can affect the state space.

Lemma 3. *Suppose that $c \in \mathrm{Com}$ is stable w.r.t. dependency specification P. Then, $\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle$ iff $\langle g(c), g(\sigma), g(s)\rangle \xrightarrow{g(\alpha)} \langle g(c'), g(\sigma'), g(s')\rangle$.*

Definition 10 (Stability for Configurations). *Let a dependency specification be given. The configuration $\langle c,\ \sigma,\ s\rangle$ is stable if whenever $\langle c,\ \sigma,\ s\rangle \Rightarrow \langle c',\ \sigma',\ s'\rangle$, then c' is a stable command.*

Theorem 1. *If $c \in \mathrm{Com}$ is flow compatible with dependency specification P for store σ and context s, and $\langle c,\ \sigma,\ s\rangle$ is stable, then c is admissible (for σ, s, P and $\sim$).*

Theorem 1 does not provide necessary conditions. In fact, there are admissible programs whose control flow *is* affected by the perturbations. However, the import of Theorem 1 is that, in order to verify Admissibility it is sufficient to check that the flow of control is not affected by the relabelling of secret inputs and of admissible outputs. Furthermore, it suffices to check this for a (smaller) subset of the reachable configurations.

To formalize this, consider a dependency specification P and an initial configuration $\langle c_0,\ \sigma_0,\ s_0\rangle$. For each configuration $\langle c,\ \sigma,\ s\rangle$ define $g(\langle c,\ \sigma,\ s\rangle)$ as the configuration that results from applying g to all three components, i.e. $g(\langle c,\ \sigma,\ s\rangle) = \langle g(c),\ g(\sigma),\ g(s)\rangle$. Then assume the existence of a set of program configurations $\{\xi_i\}_{i\in\mathcal{I}}$ where $0 \in \mathcal{I} \subseteq \mathbb{N}$, which satisfies the three properties below:

P1) $\xi_0 = \langle c_0,\ \sigma_0,\ s_0\rangle$,
P2) for all $i \in \mathcal{I}$, if $\xi_i = \langle c,\ \sigma,\ s\rangle$ then c is a stable command,
P3) for all $i \in \mathcal{I}$ and for all action α such that $\xi_i \xrightarrow{\alpha} q$, then

- there is a $j \in \mathcal{I}$ and a secret permuter g for P such that $q = g(\xi_j)$, and
- $P, s \vdash \alpha$ ok, if $\xi_i = \langle c,\ \sigma,\ s\rangle$.

Under these conditions, we can use Lemma 3 to prove the following

Theorem 2. *Consider a set $\{\xi_i\}_{i\in\mathcal{I}}$ satisfying conditions P1–P3 as above. Then, for each reachable configuration $\xi = \langle c,\ \sigma,\ s\rangle$,*

1. *there is an $i \in \mathcal{I}$ and a secret permuter g such that $\xi = g(\xi_i)$,*
2. *c is a stable command, and*
3. *if $\xi \xrightarrow{\alpha} q$ then $P, s \vdash \alpha$ ok.*

To conclude, notice that statements 2 and 3 in this theorem imply that $\langle c_0,\ \sigma_0,\ s_0\rangle$ is admissible, by means of Theorem 1. In the full version of this paper [6], we show how to apply Theorem 2 to prove that **Prog 1** (Fig. 1) is admissible for all initial stores and contexts.

8 Admissibility vs. Flow Compatibility

In general, admissibility does not imply flow compatibility. At a first glance this may seem somewhat surprising. The point, however, is that flow compatibility provides a syntactical tracing of data flow, not a semantical one. Consider for instance the command

$$\begin{array}{l}\mathrm{SECRET} := \mathbf{receive}\ a_1\,;\\ \mathbf{if}\ \mathrm{SECRET} = 0\ \mathbf{then}\ \_ := \mathbf{send}(\mathrm{SECRET}, a_2)\ \mathbf{else}\ \_ := \mathbf{send}(0, a_2)\end{array}$$

in the context of a dependency specification $P = \langle\{\mathbf{receive}\ a_1\},\ \emptyset\rangle$.

This command is clearly admissible for P (for any store and context), but not flow compatible for quite obvious reasons. However, if the control flow does not permit branching on secrets, we can show that in fact flow compatibility is implied. For this purpose some additional assumptions need to be made concerning the domains and functions involved.

Clearly, if constant functions are allowed there are trivial examples of direct flows which violate flow compatibility without necessarily violating admissibility.

However, we are able to establish the following result as a partial converse to Theorem 1.

Lemma 4. *Suppose $\langle c_0,\ \sigma_0,\ s_0\rangle$ is stable and admissible for dependency specification P. Then for all behaviours*

$$\langle c_0,\ \sigma_0,\ s_0\rangle \Rightarrow \langle c_1,\ \sigma_1,\ s_1\rangle \xrightarrow{v\,:=\,f\,w} \langle c_2,\ \sigma_2,\ s_2\rangle$$

of minimal length such that $P, s_1 \not\vdash v := f\ w$ ok, the set

$$\{[\![g(w)]\!] \mid\ g \textit{ is a secret permuter}\}$$

is finite.

Thus, if we can guarantee infinite variability of the set in Lemma 4 (which we cannot in general), flow compatibility does indeed follow from admissibility and stability.

9 Discussion and Conclusions

We have studied and presented conditions under which an implementation is guaranteed to preserve the confidentiality properties of a protocol. We first determine, using annotations, the direct flow properties. If all direct dependencies are admitted by the policy, we use an extension of the admissibility condition introduced first in [4] to detect the presence of any other dependencies. If none are detected we conclude that the implementation preserves the confidentiality properties of the protocol.

As our main results we establish close relations between the direct and the indirect dependency analysis in the case of programs which mirror the "only-high-branching-on-secrets" condition familiar from type-based information flow analyses (cf. [11, 9]). In fact, in our setting the condition is more precisely cast as "only-*permitted*-branching-on-secrets", since branching on secrets is admissible as long as its "observational content" is allowed by the dependency rules. The correspondence between the direct and the indirect dependency analysis provides an "unwinding theorem" which can be exploited to reduce a behavioral check (in our case: strong bisimulation equivalence) to an invariant.

One of the main goals of our work is to arrive at information flow analyses which can control dependencies in a secure way, rather than prevent them altogether, since this latter property prevents too many useful programs to be handled. Other attempts in this direction involve the modeling of observers as resource-bounded processes following well-established techniques in Cryptography (cf. [10]). The scope of approaches such as this remains very limited, however.

Intransitive noninterference [7] is a generalization of noninterference that admits downgrading through a trusted downgrader. Although it prevents direct downgrading (i.e. flows around the downgrader), it does not prevent Trojan Horses from exploiting legal downgrading channels to actively leak secret information. A solution is to resort to Robust Declassification [12], which provides criteria to determine whether a downgrader may be exploited by an attacker. Unfortunately, the observation powers of attackers are too strong in the presence of cryptographic functions, so that the approach cannot be applied without major changes to our examples.

One important property which our approach does not handle satisfactorily is nonce freshness. Our formalism has, as yet, no way (except by the introduction of artificial data dependencies) of introducing constraints such as "x was input after y", and thus we must at present resort to external means for this check.

One worry of more practical concern is the amount of detail needed to be provided by the dependency rules. It is quite possible that this problem can be managed in restricted contexts such as JavaCard. In general, though, it is not a priori clear how to ensure that the rules provide enough implementation freedom, nor that they are in fact correct. It may be that the rules can be produced automatically from abstract protocol and API specifications, or, alternatively, that they can be synthesized from the given implementation and then serve as input for a manual correctness check.

References

1. M. Abadi and A. D. Gordon. A Bisimulation Method for Cryptographic Protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998.
2. M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce*, May 1995.
3. E. S. Cohen. Information Transmission in Sequential Programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
4. M. Dam and P. Giambiagi. Confidentiality for Mobile Code: The case of a simple payment protocol. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, 2000.
5. R. Focardi and R. Gorrieri. A Classification of Security Properties for Process Algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
6. P. Giambiagi and M. Dam. On the Secure Implementation of Security Protocols. Full version, available from `http://www.sics.se/fdt/publications/gd03-secImpl-full.pdf`, 2003.
7. A. W. Roscoe and M. H. Goldsmith. What is Intransitive Noninterference? In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, 1999.
8. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
9. A. Sabelfeld and D. Sands. A PER Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 14(1), 2001.
10. D. Volpano. Secure Introduction of One-Way Functions. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, 2000.
11. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
12. S. Zdancewic and A. Myers. Robust Declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, 2001.