On the Secure Implementation of Security Protocols [1]

Pablo Giambiagi and Mads Dam

Swedish Institute of Computer Science, Box 1263, S-164 49 Kista, Sweden

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 theorem, reducing a behavioural correctness check (strong bisimulation) to an invariant.

Key words: Semantics-based security, confidentiality, information flow, protocol implementation, admissibility, security policy

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

Email addresses: `pablo@sics.se` (Pablo Giambiagi), `mfd@sics.se` (Mads Dam).

[1] 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.

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 are 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, even without data-dependent branching.

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

$$\mathbf{send}(v, \mathit{OUT}) \leftarrow k \lhd \mathbf{key}\, \mathit{BOB},\ m \lhd \mathbf{get}\, \mathit{IN},\ v \lhd \mathbf{enc}(m, k)$$

indicating that, if upon its last invocation of the **get** method with argument IN the protocol received m (and analogously for **key**, BOB, and k), then the next invocation of **send** with second parameter OUT must, as its first parameter, receive the encryption of m with key k.

A dependency specification defines an information flow property by characterizing the set of allowed flows. For example, the rule above allows the flow of the secret plaintext m to public channel OUT only if m is first encrypted using BOB's key. 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 [1]. 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.

It is worth pointing out that the approach presented here, like any semantics-based characterization of information flow, can only deal with flows that are observable within the semantic model. Covert channels may exist that exploit the unavoidable gap between model and reality. For example, probabilistic and timing covert channels may run on undetected if probabilistic, resp. timing, aspects are not modeled. Even when the examples in this paper assume a possibilistic and untimed semantic model, this is not a prerequisite of the approach. The idea of invariance under perturbation is parametric in, precisely, the definition of invariance (i.e. $\sim$).

The paper has two main contributions. First, we show how the idea can be realized in the context of a simple sequential imperative language, IMP. 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. [2]). The idea of invariance under perturbation and logical relations underpins most work on secrecy and information flow theory (see [3]), though not always very explicitly (cf. [4–7]). The main point, in contrast e.g. to work by Volpano [8], 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 IMP and introduce the main example used in the paper, a declassifier which will leak a secret provided it has been authorized to do so by some external agent. In Section 3 we introduce an annotated semantics, used in Section 4 to formalise 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

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

Fig. 1. IMP Syntax

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 IMP, 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.

Figure 1 defines the syntax of IMP, with variables $x \in \mathit{Var}$, including the anonymous variable $\_$, primitive function and procedure calls, and primitive data types including natural numbers ($n \in \mathit{Nat}$) and channels ($a \in \mathit{Chan}$). The set of primitive function symbols, ranged over by pf, includes the standard arithmetic and logical operators as well as tuple projectors ($\pi_{\mathbf{i}}$). Each primitive function $pf \colon \mathit{Val} \to \mathit{Val}$ is assumed to satisfy $pf(\textbf{xcpt}) = \textbf{xcpt}$ (i.e. primitive function invocations propagate exceptions from arguments to results). Moreover, primitive functions are assumed to execute in constant time, regardless of their arguments, and to have no side effects. Communication effects are brought out using transition labels in the next section. There are also non-primitive (or API) functions, ranged over by h, for (nondeterministic and non-malleable) encryption (**enc**), decryption (**dec**), extracting a key from a keystore (**key**), and receiving resp. sending a value on a channel (**get** and **send**). Considering the commands of the language, $*$ represents the *empty command* that is used only to ease the presentation of the semantics. It is assumed that $*$ satisfies the following structural congruences: $*; c \equiv c; * \equiv c$ and $\textbf{try } * \textbf{ catch } c \equiv *$. Observe that, while **skip** can be executed in one step, $*$ is not to be executed at all. While exceptions do not enhance in any fundamental way the expressiveness of the language, a simple exception mechanism helps approximate the way cryptographic protocols are coded in real-life imperative

Message 1	$C \to DCL$	:	$\{secret\}_{K_{C,D}}$	
Message 2	$DCL \to C$	:	$\{decl, secret\}_{K_{C,D}}$	
Message 3	$C \to PUBLIC$	:	$secret$	if $decl = YES$

Fig. 2. A Declassification Protocol

programming languages. Furthermore, the possibility of exceptions introduces unobvious control branching points which, in turn, may induce implicit information flows.

As a running example we use a simple declassifier, representative of applications which are required to input a collection of data, some sensitive, some not, perform some cryptographic operations on the data, and occasionally transmit the results on a public channel.

2.1 A Declassification Protocol

This protocol involves three agents, a client (C), a declassifier (DCL), and the public ($PUBLIC$). The client is interested in declassifying a secret to the public, but this needs the approval of the declassifier. Following the protocol, the client sends the secret to the declassifier for consideration; the declassifier examines it and decides whether to approve or not the request. The declassifier's response, *decl*, is sent back to the client together with the secret. The client can then communicate the secret to the public only if the response was affirmative (i.e. $decl = YES$). A description of the protocol using the standard (and informal) notation appears in Figure 2.

According to the protocol, all communication between the client and the declassifier is to be encrypted using a key, $K_{C,D}$ shared between the client and the declassifier. It is also assumed that the secret contains enough information so that the declassifier can identify it properly.

Figure 3 shows what a simple implementation of the Client's side of the declassification protocol might look like in IMP. Keywords in uppercase letters denote constant variables, e.g. SECRET identifies the channel that provides the secret data to the client. 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

```
1:  $key$ := key DCL ;
2:  while true do
3:    $secret$ := get SECRET ;
4:    $outPacket$ := enc($secret$, $key$);
5:    _ := send($outPacket$, DCL) ;
6:    $encResp$ := get DCL ;
7:    try
8:      $resp$ := dec($encResp$, $key$) ;
9:      if $\pi_1(resp)$ = YES and $\pi_2(resp) = secret$
10:     then _ := send($secret$, PUBLIC)
11:     else skip
12:   catch skip
13: end
```

Fig. 3. Client - sample implementation

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 "forget" to fully verify the Declassifier's response by replacing line 9 of Figure 3 with

```
9:        if $\pi_2(resp) = secret$ then
```

Essentially, this would declassify the secret even when the Declassifier may explicitly forbid the operation. The implementation might also try to replace good keys by bad ones, for instance by replacing line 1 with

```
1:  $key$ := key ATTACKER ;
```

In the following example, the execution of line 8.2 may look innocuous by itself, but in the context of the conditional statement it creates an indirect leak of secret information:

```
8.1:      if $secret$ = FIXEDVAL then
8.2:        _ := send(DUMMY, PUBLIC)
8.3:      else skip ;
```

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 secret and critical data (an example of the latter is a public key needed to encrypt a secret before communication on a public channel. Since a malicious implementation could simply use a different key for encryption, it is also necessary to track its origins). Once this is accomplished, other techniques 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 **get** 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 IMP 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

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

is intended to indicate that the value 347 was computed by applying the API function **enc** to the pair $(717, 101)$ for which the left hand component was computed by evaluating **get** a, and so on.

Annotated expressions and values are obtained by changing the definition of expressions (resp. values) in Figure 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	$::=$	$f\,w$

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

Table 1 defines the small-step semantics for expression evaluation. The transition relation has the shape

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

Table 1
Annotated semantics, expressions

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

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

where α is an action of the form τ (internal computation step) or $v \lhd h\,w$ (h is applied to the annotated value w resulting in the value v), and σ is an annotated store, a partial function $\sigma \in \mathit{aStore} \triangleq [\mathit{Var} \to \mathit{aBVal}]$. In particular notice that internal computation corresponds either to evaluation of primitive function calls or to the propagation of exceptions out of tuples.

Annotated values give only static information in the style "the value v' was computed by evaluating $\mathbf{key}(v : \mathbf{get}\ a)$", but no information concerning which actual invocations of the **key** and **get** functions were involved. However, our *dependency specifications* (introduced on p. 2 and formalised by Def. 3) require that we can identify value annotations that correspond to last invocations. For that purpose 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 $s : [\mathit{Ann} \to \mathit{Val}]$.

So, if s is a context then $s\,\varphi$ is the last value returned by the annotated function call φ. Observe that it is the semantics of the *dependency specification* what determines the contextual (or historical) information that should be collected. Richer specification languages than the one used in this paper could certainly require more contextual information (e.g. the result of all function invocations, their relative order or even the moment in time in which each event took place).

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, $\sigma \in$ *aStore* and $s \in$ *Context*. We use $\langle \sigma,\ s\rangle$ as abbreviation of $\langle *,\ \sigma,\ s\rangle$.

The annotated command-level semantics is shown in Table 2. We use $\sigma[\beta/x]$ to denote store update (i.e. $\sigma[\beta/x](x) = \beta$ and $\sigma[\beta/x](y) = \sigma(y)$ for $y \neq x$). The same notation is applied for context updates. To lift transitions between expressions to transitions between configurations, we use reduction contexts $r[\cdot] ::= x := \cdot \mid \mathbf{if}\ \cdot\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$.

Table 2
Small-step semantics for annotated commands

$$\vdash \langle \mathbf{skip},\ \sigma,\ s\rangle \xrightarrow{\tau} \langle \sigma, s\rangle \qquad \vdash \langle x := \beta,\ \sigma,\ s\rangle \xrightarrow{\tau} \langle \sigma[\beta/x], s\rangle$$

$$\frac{\vdash \langle c_0,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0',\ \sigma',\ s'\rangle}{\vdash \langle c_0; c_1,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0'; c_1,\ \sigma',\ s'\rangle} \qquad \frac{[\![\beta]\!] = \mathbf{true}}{\vdash \langle \mathbf{if}\ \beta\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1,\ \sigma,\ s\rangle \xrightarrow{\tau} \langle c_0,\ \sigma,\ s\rangle}$$

$$\frac{[\![\beta]\!] \neq \mathbf{true}}{\vdash \langle \mathbf{if}\ \beta\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1,\ \sigma,\ s\rangle \xrightarrow{\tau} \langle c_1,\ \sigma,\ s\rangle}$$

$$\vdash \langle \mathbf{while}\ \epsilon\ \mathbf{do}\ c\ \mathbf{end},\ \sigma,\ s\rangle \xrightarrow{\tau} \langle \mathbf{if}\ \epsilon\ \mathbf{then}\ c; \mathbf{while}\ \epsilon\ \mathbf{do}\ c\ \mathbf{end}\ \mathbf{else}\ \mathbf{skip},\ \sigma,\ s\rangle$$

$$\frac{\vdash \langle c_0,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0',\ \sigma',\ s'\rangle}{\vdash \langle \mathbf{try}\ c_0\ \mathbf{catch}\ c_1,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle \mathbf{try}\ c_0'\ \mathbf{catch}\ c_1,\ \sigma',\ s'\rangle}$$

$$\vdash \langle \mathbf{try}\ \mathbf{throw}\ \mathbf{catch}\ c_1,\ \sigma,\ s\rangle \xrightarrow{\tau} \langle c_1,\ \sigma,\ s\rangle$$

$$\frac{\sigma \vdash \epsilon \xrightarrow{\tau} \epsilon'}{\vdash \langle r[\epsilon],\ \sigma,\ s\rangle \xrightarrow{\tau} \langle r[\epsilon'],\ \sigma,\ s\rangle} \qquad \frac{\sigma \vdash \epsilon \xrightarrow{v \lhd \varphi} \epsilon'}{\vdash \langle r[\epsilon], \sigma, s\rangle \xrightarrow{v \lhd \varphi} \langle r[\epsilon'], \sigma, s[v/\varphi]\rangle}$$

$$\frac{[\![w]\!] = \mathbf{xcpt}}{\vdash \langle r[w],\ \sigma,\ s\rangle \xrightarrow{\tau} \langle \mathbf{throw},\ \sigma,\ s\rangle}$$

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 formalise the permitted, direct flows.

Definition 3 (Dependency Specification) A *dependency specification* is a pair $P = \langle \mathcal{H}, \mathcal{A}\rangle$ where $\mathcal{H} \subseteq \mathit{Ann}$ is a set of annotations of the form $h\,w$, and $\mathcal{A}$ is a finite set of rules of the form

$$h\,e \leftarrow x_1 \lhd h_1\,e_1,\ \ldots,\ x_n \lhd h_n\,e_n\ \mathit{when}\ \psi \qquad (1)$$

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

$$
\begin{array}{lcl}
eval(w) & = & [\![w]\!] \\
eval(\psi_1,\ldots,\psi_n) & = & \begin{cases} (eval(\psi_1),\ldots,eval(\psi_n)) & \text{if } \forall i.\ eval(\psi_i) \neq \mathbf{xcpt} \\ \mathbf{xcpt} & \text{otherwise} \end{cases} \\
eval(pf\ \psi) & = & pf(eval(\psi)) \\
eval(\psi_1 \equiv \psi_2) & = & (eval(\psi_1) = eval(\psi_2))
\end{array}
$$

Fig. 4. Evaluation of boolean conditions in dependency rules

free variables in $\{x_1,\ldots x_n\}$ and syntax given by

$$\text{Boolean conditions } (\Psi) \quad \psi \ ::= \ w \mid x \mid (\psi_1,\ldots,\psi_n) \mid pf\ \psi \mid \psi_1 \equiv \psi_2 \ .$$

The intention is that $\mathcal{H}$ represents a set of secret entry points (such as: **get** *SECRET*), and that the rules in $\mathcal{A}$ determine a bound to the secret data flows that are allowed in an implementation.

A rule in the specification declares an API function invocation $h\,e$ to be admissible if the conditions to the right of the arrow are satisfied. Informally, conditions of the form $x_i \lhd h_i\ e_i$ are satisfied if variable x_i matches the last value returned by $h_i\,e_i$. Restricting attention to last invocations helps keep specification rules simple, a decision motivated by the fact that most cryptographic protocol sessions involve a reduced number of (mostly distinct) API invocations. More importantly, notice that dependency specifications cannot allow/reject flows on the basis of temporal constraints such as the order of function calls.

The boolean expression ψ represents an extra condition that relates the values returned by the different API function invocations, and that may be evaluated using function $eval\colon \Psi \rightarrow Val$ (see Fig. 4). Recall that primitive functions pf include the standard arithmetic and logical operators. Moreover, since pf is required to preserve exceptions we obtain that $\mathbf{xcpt} \equiv \mathbf{xcpt}$ evaluates to **true**, but $\mathbf{xcpt} = \mathbf{xcpt}$ evaluates to **xcpt**.

To formalise the semantics of a dependency specification, we need one more definition. Let a context s be given. A *valid substitution* for rule (1) is an annotated store $\rho \in aStore$ such that

(1) $\rho(x_i) = s(h_i\,(e_i\rho))\colon h_i\,(e_i\rho)$ for all $i : 1 \leq i \leq n$,
(2) if x does not appear in $x_i \lhd h_i\,e_i$ ($\forall i.\, 1 \leq i \leq n$), then $\rho(x)$ has no annotation in $\mathcal{H}$,
(3) $eval(\psi\rho) = \mathbf{true}$.

That is, the value bound to x_i by ρ should be equal to the last value returned by the annotated function call $h_i\,(e_i\rho)$; all variables that are not thus determined by the context s must not have secret annotations; and the boolean condition should be satisfied. By $e\rho$ we mean the annotated ground expression ($aExpr$) that results from substituting $\rho(x)$ for every variable x in e. It is easy to check that the restrictions on e (resp. e_i) in Def. 3 guarantee that $e\rho$ (resp. $e_i\rho$) is an annotated (basic) value. Notation: When $eval(\psi\rho) = \mathbf{true}$ regardless of the substitution ρ, we usually drop ψ and the preceding keyword **when** from the dependency rule (1).

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 $v \lhd h\,w$. A dependency specification $P = \langle \mathcal{H}, \mathcal{A} \rangle$ *admits annotated action* α in context s iff either

(1) no annotation in w belongs to $\mathcal{H}$ (that is, the actual parameters do not directly depend on any secret), or
(2) there is a rule $h\,e \leftarrow x_1 \lhd h_1\,e_1,\ \ldots,\ x_n \lhd h_n\,e_n\ \mathit{when}\ \psi$ in $\mathcal{A}$ and a valid substitution ρ for this rule such that $e\rho = w$.

If any one of these conditions is satisfied we write $P, s \vdash \alpha\ \mathit{ok}$.

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. $P, s \vdash \tau\ \mathit{ok}$). Notice as well that the definition of $P, s \vdash v \lhd h\,w\ \mathit{ok}$ is independent of the value of v.

Example 5 (Dependency Specification for the Declassifier's Client) For the declassification protocol, the only pieces of information that the Client should protect are the secret itself (obtained through channel SECRET, and all the keys it shares with other principals. Therefore, $\mathcal{H} = \{\mathbf{get}\ \mathit{SECRET}\} \cup \{\mathbf{key}\ w \mid w \in a\mathit{Val}\}$, and $\mathcal{A}$ consists of rules for encryption, for decryption, for sending encrypted packets to the declassifier, and for declassifying the secret.

$$\begin{array}{rcll}
\mathbf{enc}(s,k) & \leftarrow & s \lhd \mathbf{get}\ \mathit{SECRET},\ \ k \lhd \mathbf{key}\ \mathit{DCL} & (2)\\
\mathbf{send}(o,\mathit{DCL}) & \leftarrow & o \lhd \mathbf{enc}\,x & (3)\\
\mathbf{dec}(m,k) & \leftarrow & m \lhd \mathbf{get}\ \mathit{DCL},\ \ k \lhd \mathbf{key}\ \mathit{DCL} & (4)\\
\mathbf{send}(s,\mathit{PUBLIC}) & \leftarrow & s \lhd \mathbf{get}\ \mathit{SECRET},\ \ r \lhd \mathbf{dec}(m,k) & (5)\\
 & \mathit{when} & \pi_1(r) = \mathit{YES} \wedge \pi_2(r) = s &
\end{array}$$

According to rule (2), each time the encryption function (**enc**) is invoked with some sensitive parameter (i.e. a value with an annotation in $\mathcal{H}$), it must be the case that the plaintext is the secret which was received last (by means of function **get**), and that the encryption key is the Declassifier's. By rule (3) all sensitive information sent to the Declassifier must be encrypted. There is no need to further restrict the argument (i.e. x) passed to **enc**, for that is already constrained by rule (2). Since **key** $DCL \in \mathcal{H}$, rule (4) is needed to admit the decryption operation implicit in the processing of Message 2 (cf. Fig. 2). Finally, by rule (5) a secret may be declassified provided the decrypted message both says so and correctly matches the current secret. $\square$

As the example shows, 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 6 Let the dependency specification $P = \langle \mathcal{H}, \mathcal{A}\rangle$ 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 7 (Flow Compatibility for the Declassifier's Client)
The program of Figure 3 is flow compatible with the Declassifier's Client dependency specification of Example 5, for any initial store σ. This is seen by proving that appropriate invariants hold each time execution reaches one of

the statements in lines 4, 5, 8 and 10. For example, at line 10 the store σ and context s can be shown to always satisfy, for suitable choices of $v_1, \ldots, v_4$,

$$\begin{array}{ll}
\sigma(\mathit{secret}) = v_1 : \textbf{get}\ \mathit{SECRET} & \sigma(\mathit{key}) = v_2 : \textbf{key}\ \mathit{DCL} \\
\sigma(\mathit{encResp}) = v_3 : \textbf{get}\ \mathit{DCL} & s(\textbf{dec}(\sigma(\mathit{encResp}), \sigma(\mathit{key})) = v_4 \\
\pi_1(v_4) = \mathit{YES} & \pi_2(v_5) = v_1
\end{array}$$

It is then a simple matter to check $P, s \vdash \textbf{send}(v_1 : \textbf{get}\ \mathit{SECRET},\ \mathit{PUBLIC})\ \mathit{ok}$ using rule (5) in Example 5 and the following valid substitution ρ:

$$\begin{array}{ll}
\rho(s) = \sigma(\mathit{secret}) & \rho(k) = \sigma(\mathit{key}) \\
\rho(m) = \sigma(\mathit{encResp}) & \rho(r) = v_4 : \textbf{dec}(\rho(m), \rho(k))
\end{array}$$

Consider now the three malicious implementations discussed by the end of Section 2.1. In the first two cases, flow compatibility is violated, as expected. On the other hand, the third malicious implementation *is* flow compatible, also as expected, since the indirect leak will not be traced by the annotation regime. $\square$

6 Admissibility

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

In the context of multilevel security it is by now quite well understood how to model absence of information flow (from a high security level –or *clearance*– to a lower one) as invariance of system behaviour under perturbation of secret inputs (cf. [4–7], see also [9] 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 level 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 8 (Value Permutation) A bijection $g\colon aVal \rightarrow aVal$ is a *value permutation* if it preserves the structure of annotated values:

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

and the meaning of primitive functions:

(4) $v = pf(\llbracket w \rrbracket)$ iff $v' = pf(\llbracket w' \rrbracket)$, given $g(v : pf\ w) = v' : pf\ w'$.

We do not require the meaning of non-primitive functions to be preserved by value permutations. Since we do not want to prescribe any particular behaviour for API implementations, our semantics does not actually assign a meaning to non-primitive functions in the first place, so formally there is nothing to preserve. This is so, since API implementations in our framework are trusted: Dependency specifications simply state under what conditions an API function may be invoked, regardless of how it is actually implemented.

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

$$g(s)(f\ w) \triangleq \llbracket g(s(f\,w') : f\,w') \rrbracket\,,\ \text{where}\ w' = g^{-1}(w)\,. \tag{6}$$

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 API function φ, resp. $g(\varphi)$.

Lemma 9 *If* $g(v \lhd \varphi) = v' \lhd \varphi'$ *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 10 (Secret Permuter) Assume given a dependency specification $P = (\mathcal{H}, \mathcal{A})$. A *secret permuter for* P is a value permutation g satisfying the following conditions:

(1) if w does not contain annotations in $\mathcal{H}$ then $g(w) = w$,
(2) ρ is a valid substitution for rule $r \in \mathcal{A}$ and context s if and only if $\rho'(x) \triangleq g(\rho(x))$ is a valid substitution for rule r and context $g(s)$,
(3) $g(\mathbf{xcpt} : h\, w) = \mathbf{xcpt} : h\, g(w)$, for every API function h,

As expected, a secret permuter affects only secret values. This is implied by the first condition in Definition 10. Condition (10.2) implies that a secret permuter must respect the restrictions imposed by the boolean conditions in each dependency rule. On the other hand, we assume that the exceptional behaviour of an API function is always observable. Thus, if the execution of $h\ w$ raises an exception, we should not consider those cases where $h\ g(w)$ does not raise an exception. This is reflected by condition (10.3).

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

Lemma 11 *Let* g *be a secret permuter. Then*

(1) $g^{-1}(g(s)) = s$,

(2) g^{-1} *is a secret permuter, and*

(3) if $P, s \vdash \alpha$ *ok then* $P, g(s) \vdash g(\alpha)$ *ok*.

Proposition 12 (Composition of Secret Permuters) *Given a dependency specification, the set of secret permuters is closed under functional composition.*

Example 13 (Secret Permuter for the Declassification Example) Here we give an example of a secret permuter for the dependency specification in Example 5. First we verify that it is a value permutation (Def. 8), and then that it is indeed a secret permuter (Def. 10).

Let $g\colon aVal \rightarrow aVal$ exchange annotated values as follows:

$$v_1 : \mathbf{get}\ \mathit{SECRET} \leftrightarrow v_1' : \mathbf{get}\ \mathit{SECRET}$$
$$v_2 : \mathbf{key}\ \mathit{DCL} \leftrightarrow v_2' : \mathbf{key}\ \mathit{DCL}$$
$$v_4 : \mathbf{dec}(v_3 : \mathbf{get}\ \mathit{DCL}, v_2 : \mathbf{key}\ \mathit{DCL}) \leftrightarrow$$
$$v_4' : \mathbf{dec}(v_3 : \mathbf{get}\ \mathit{DCL}, v_2' : \mathbf{key}\ \mathit{DCL})$$

$$v_1 : \pi_2(v_4 : \mathbf{dec}(v_3 : \mathbf{get}\ DCL, v_2 : \mathbf{key}\ DCL)) \leftrightarrow$$
$$v_1' : \pi_2(v_4' : \mathbf{dec}(v_3 : \mathbf{get}\ DCL, v_2' : \mathbf{key}\ DCL))$$

for some fixed basic values v_i and v_j' satisfying $v_i \neq v_i'$, $\pi_1(v_4) = \pi_1(v_4') = YES$, $\pi_2(v_4) = v_1$ and $\pi_2(v_4') = v_1'$. On all other values, g acts in accordance with conditions in Defs. 8 and 10. Conditions (8.1)–(8.4), (10.1) and (10.3) are easily validated. To verify condition (10.2) consider rule (5) in the dependency specification of the Declassifier's Client (Ex. 5). Given a context s such that:

$$s(\mathbf{get}\ SECRET) = v_1 \qquad s(\mathbf{dec}(v_3 : \mathbf{get}\ DCL,\ v_2 : \mathbf{key}\ DCL)) = v_4$$
$$s(\mathbf{key}\ DCL) = v_2$$

assume ρ is a valid substitution for s and the mentioned rule. Let $\rho'(x) \triangleq g(\rho(x))$. We need to show that ρ' satisfies conditions (1)-(3) in the definition of valid substitution within context $g(s)$. Condition (2) holds trivially. Let us illustrate the verification of condition (1) with variable s in rule (5).

$$\begin{aligned}\rho'(s) &= g(\rho(s)) = g(s(\mathbf{get}\ SECRET) : \mathbf{get}\ SECRET) \\ &= g(v_1 : \mathbf{get}\ SECRET) = [\![g(v_1 : \mathbf{get}\ SECRET)]\!] : \mathbf{get}\ SECRET \\ &= g(s)(\mathbf{get}\ SECRET) : \mathbf{get}\ SECRET\end{aligned}$$

Notice that $[\![\rho'(s)]\!] = v_1'$. In the same manner it can be checked that $[\![\rho'(n)]\!] = [\![\rho(n)]\!]$ and $[\![\rho'(r)]\!] = v_4'$, which is enough to verify condition (3) in the definition of valid substitution. Therefore g is a secret permuter for the Client in our declassification protocol. $\square$

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, $g(r := \mathbf{dec}(b : \mathbf{get}\ DCL, b' : \mathbf{key}\ DCL)) = r := \mathbf{dec}(b : \mathbf{get}\ DCL, g(b' : \mathbf{key}\ DCL))$. 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

$$\frac{\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle}{\langle c[g],\ \sigma,\ s\rangle \xrightarrow{[\![g(s,\alpha)]\!]} \langle c'[g], \sigma', s'\rangle} \tag{7}$$

where $[\![v \lhd h\, w]\!] = v \lhd h\, [\![w]\!]$, 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 (7). 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 14 (Admissibility) A command $c \in Com$ is admissible for the store σ and context s, the dependency specification P, if for all secret permuters g for P:

$$\langle c[I], \sigma, s\rangle \sim \langle g(c)[g^{-1}], g(\sigma), g(s)\rangle \tag{8}$$

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.

Example 15 Recall the definition of secret permuter g for the Declassifier's Client in Example 13. If we modified only slightly our assumptions so that $\pi_2(v_4') = \mathbf{xcpt}$ and $\pi_1(v_4) \notin \{YES, \mathbf{xcpt}\}$, we could define a secret permuter g' just as we did with g. This means that the dependency specification of Example 5 expects the behaviour of a Client implementation to be invariant under, among other permuters, g'. This is not true of the implementation shown in Figure 3. There is one possible trace where control branches from line 9 to line 11 (if $[\![\sigma(\mathit{resp})]\!] = v_4$) and another where it branches to line 12 (if $[\![\sigma(\mathit{resp})]\!] = v_4'$). In other words, the occurrence of an exception in executing line 9 may leak inadmissible information about the relation between *resp* and *secret*. If this information were considered innocuous, we could simply add one more rule to the dependency specification:

$$\begin{aligned} * \quad \leftarrow \quad & s \lhd \mathbf{get}\ \mathit{SECRET},\ r \lhd \mathbf{dec}(m,k) \\ \mathit{when}\ \ & \pi_1(r) = \mathbf{xcpt} \vee \pi_2(r) = \mathbf{xcpt} \end{aligned} \tag{9}$$

where $*$ represents some arbitrary $h\,e$ where e does not mention any variable on the right-hand side of the rule. Its purpose is not to declare admissible certain invocations of the function h (h can always be invoked with non-sensitive arguments) but to introduce a constraint on the set of secret permuters. Observe that, in the presence of rule (9), g' is no longer a valid secret permuter for the extended dependency specification. $\square$

7 Local Verification Conditions

Applying the definition of admissibility out of the box can be quite arduous, since it is tantamount to searching for, and checking, a bisimulation relation for each possible secret permuter. 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 16 (Stability for Commands) Let a dependency specification P be given. Let $\preccurlyeq$ be the smallest reflexive and transitive relation over commands such that $c_0 \preccurlyeq c_0; c_1$ and $c_0 \preccurlyeq \mathbf{try}\ c_0\ \mathbf{catch}\ c_1$, for all commands c_0 and c_1. The command $c \in \mathit{Com}$ is *stable* if for all $c' \preccurlyeq c$ and for all secret permuters g,

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

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 17 *Suppose that $c \in \mathit{Com}$ 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 18 (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 19 *If $c \in Com$ 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 19 does not provide necessary conditions. In fact, there are admissible programs whose control flow *is* affected by the perturbations. However, the import of Theorem 19 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 formalise 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 following properties:

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 actions α such that $\xi_i \xrightarrow{\alpha} q$, then

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

Under these conditions, we obtain our main verification result with the aid of Lemma 17 and Theorem 19:

Theorem 20 *Let P be a dependency specification and $\langle c_0, \sigma_0, s_0\rangle$ an initial configuration. If there is a set of configurations $\{\xi_i\}_{\mathcal{I}}$ satisfying P1–P3, then c_0 is admissible (for σ_0, s_0, P and $\sim$).*

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

$SECRET :=$ **get** a_1 ;

$$\textbf{if } \mathit{SECRET} = 0 \textbf{ then } \_ := \textbf{send}(\mathit{SECRET}, a_2) \textbf{ else } \_ := \textbf{send}(0, a_2)$$

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

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 19.

Lemma 21 *Suppose $\langle c_0,\ \sigma_0,\ s_0 \rangle$ is stable and admissible for dependency specification P. Then for all behaviours*

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

of minimal length such that $P, s_1 \not\vdash v \lhd h\, w$ ok, the set

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

is finite.

Thus, if we can guarantee infinite variability of the set in Lemma 21 (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 [1] 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. [6,7]). 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 behavioural check (in our case: strong bisimulation equivalence) to an invariant.

The notion of admissibility has relations to representation independence. The latter concerns the problem of showing, for a given program, that its behaviour depends only on abstract values, not on details concerning their concrete representation. This is typically handled using logical relations (cf. [10]). The task is to give a relation that describes how different concrete representations implement the same abstraction, and to show that if all methods preserve this relation then it will be preserved by any client program. If one can show that, along a suitably defined external interface, the relation is the identity, then one can conclude that no client program which respects this interface can leak representation dependent information. The analogy with e.g. the PER model of Sabelfeld and Sands [11] is clear, except that, in the case of representation independence, finer variability at base types is required. On the other hand, variability across the external interface is prohibited (or, in the case of data refinement [12], required to preserve the implementation ordering), and one can in fact view the present paper as offering one scenario, and possible approach, for lifting this restriction.

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 excludes too many useful programs. Other attempts in this direction involve the modeling of observers as resource-bounded processes following well-established techniques in Cryptography (cf. [8,13]). The scope of approaches such as these remains very limited, however.

Intransitive noninterference [14] 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 [15], which provides criteria to determine whether a downgrader may be exploited by an attacker. Unfortunately, this technique considers attackers whose observation power turns out to be too strong in the presence of cryptographic functions, so that the approach cannot be applied without major changes to our examples.

Dependency specifications are abstract in the sense that they do not request compliance with many functional properties of the security protocol. For example, the Client specification (Example 5) does not prevent an implementation from submitting the same secret, over and over, to the Declassifier. This is quite safe, as we assure that aspects like the number of retransmissions, or their timing properties, cannot be used to create covert channels.

However, there are occasions in which compliance with functional behaviour is critical. In particular, 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 expressing 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.

A Proofs

Lemma 9 *If $g(v \lhd \varphi) = v' \lhd \varphi'$ then $g(s[v/\varphi]) = g(s)[v'/\varphi']$.*

PROOF. Let $\varphi = f\,w$ and $\varphi' = f\,w'$ where $w' = g(w)$. We need to show that $g(s[v/f\ w]) = g(s)[v'/f\ w']$. Let $f_1\ w_1 \in Ann$. Assume first $f_1\ w_1 = f\ w'$. We compute:

$$\begin{aligned} g(s[v/f\ w])(f\ w') &= g(s[v/f\ w])(f\ g(w)) \\ &= v' && \text{(by (6))} \\ &= g(s)[v'/f\ w'](f\ w') \end{aligned}$$

In case $f_1\ w_1 \neq f\ w'$, let $v'' \triangleq s[v/f\ w](f_1\ g^{-1}(w_1)) = s(f_1\ g^{-1}(w_1))$. We obtain:

$$\begin{aligned} g(s[v/f\ w])(f_1\ w_1) &= [\![g(v'' : f_1\ g^{-1}(w_1))]\!] \\ &= g(s)(f_1\ w_1) \\ &= g(s)[v'/f\ w'](f_1\ w_1). \qquad \square \end{aligned}$$

Lemma 11 *Let g be a secret permuter. Then*

(1) $g^{-1}(g(s)) = s$ *,*
(2) g^{-1} *is a secret permuter, and*
(3) if $P, s \vdash \alpha$ *ok then* $P, g(s) \vdash g(\alpha)$ *ok .*

PROOF.

(1) Notice first that (6) is equivalent to

$$g(s)(f\,w) : f\,w = g(\,s(f\,g^{-1}(w)) : f\,g^{-1}(w)\,) \tag{A.1}$$

Given $f\,w \in \mathit{Ann}$, let $w' \triangleq g(w)$. Then

$$\begin{aligned} g^{-1}(g(s))(f\,w) : f\,w &= g^{-1}(\,g(s)(f\,w') : f\,w'\,) && \text{(by (A.1))} \\ &= g^{-1}(\,g(s(f\,w) : f\,w)\,) && \text{(by (A.1))} \\ &= s(f\,w) : f\,w \end{aligned}$$

Therefore, $g^{-1}(g(s))(f\,w) = s(f\,w)$ for all $f\,w \in \mathit{Ann}$.

(2) This is a simple check, using that g, when extended over contexts, is a bijection (a consequence of item (1)).

(3) Let $P = \langle \mathcal{H}, \mathcal{A} \rangle$. Assume $\alpha = v \lhd h\,w$ where w has (at least) an annotation in $\mathcal{H}$ (all other cases are trivial). Since $P, s \vdash \alpha\ \mathit{ok}$, there must be a rule $h\,e \leftarrow x_1 \lhd h_1\,e_1,\ \ldots,\ x_n \lhd h_n\,e_n\ \mathit{when}\ \psi$ in $\mathcal{A}$ and a valid substitution ρ for this rule such that $e\rho = w$ (Def. 4).

By Def. 10.2, $\rho'(x) \triangleq g(\rho(x))$ is also a valid substitution for the rule above. Assume $g(v : h\,w) = v' : h\,w'$. It then only remains to notice that, by induction on the structure of e, $e\rho' = g(e\rho) = w'$ and therefore $P, g(s) \vdash g(\alpha)\ \mathit{ok}$. □

Corollary A.1 *Let g be a secret permuter for dependency specification P. If* $P, s \vdash \alpha$ *ok then* $g^{-1}(g(s), g(\alpha)) = \alpha$ *.*

PROOF.

$$\begin{aligned} g^{-1}(g(s), g(\alpha)) &= g^{-1}(g(\alpha)) \quad \text{(by Lemma 11.3, } P, g(s) \vdash g(\alpha)\ \mathit{ok}\text{)} \\ &= \alpha \end{aligned}$$
□

Proposition 12 (Composition of Secret Permuters) *Given a dependency specification, the set of secret permuters is closed under functional composition.*

PROOF. Let g and g' be two secret permuters. We need to prove that $t = g \circ g'$ is a secret permuter too (where $g \circ g'(x) \triangleq g(g'(x))$). That t is a value permutation (Def. 8) is straightforward. It is also immediate that t satisfies (10.3).

The key observation needed to check that t satisfies (10.1) is that if $f\ w$ contains no annotation in $\mathcal{H}$ then neither does $f\, g'(w)$. This is so because w cannot have an annotation in $\mathcal{H}$ and, since g' is a secret permuter, $g'(w) = w$ (also by (10.1)).

Finally, for condition (10.2) it is enough to check that $t(s) = g(g'(s))$ (observe that here t is a function over contexts, so the equality is not immediate from the definition of t as a function over annotated values). Given $f\, w \in Ann$, let $w' \triangleq g^{-1}(w)$ and $w'' = g'^{-1}(w') = t^{-1}(w)$. Then

$$\begin{aligned}
g(g'(s))(f\,w) : f\,w &= g(\,g'(s)(f\,w') : f\,w'\,) && \text{(by (A.1))}\\
&= g(\,g'(s(f\,w'') : f\,w'')\,) && \text{(by (A.1))}\\
&= t(s(f\,t^{-1}(w)) : f\,t^{-1}(w))\,) &&\\
&= t(s)(f\,w) : f\,w && \text{(by (A.1))}
\end{aligned}$$

Therefore, $t(s)(f\,w) = g(g'(s))(f\,w)$ for all $f\,w \in Ann$. $\square$

Remark A.2 *For a fixed dependency specification, if $c' \in Com$ is stable and $c \preccurlyeq c'$, then c is stable.*

Lemma A.3 *Let a dependency specification be given. Then, for all secret permuters g, $c \in Com$ is stable iff $g(c)$ is stable.*

PROOF. Follows from Prop. 12. $\square$

Lemma A.4 *Suppose that $r[\epsilon_0]$ is a stable command and ϵ is a subterm of ϵ_0. If $\sigma \vdash \epsilon \xrightarrow{\alpha} \epsilon'$ then $g(\sigma) \vdash g(\epsilon) \xrightarrow{g(\alpha)} g(\epsilon')$.*

PROOF. The proof proceeds by induction on the structure of derivations of

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

- Case $\epsilon = x$:
  Since $\epsilon' = \sigma(x)$, $\alpha = \tau$, $g(x) = x$, and $g(\sigma) \vdash x \xrightarrow{\tau} g(\sigma)(x)$, it is immediate that $g(\sigma) \vdash g(x) \xrightarrow{g(\alpha)} g(\sigma(x))$.
- Case $\epsilon = (\ldots, \epsilon_1, \ldots)$ and $\sigma \vdash \epsilon_1 \xrightarrow{\alpha} \epsilon_1'$:
  From the annotated semantics for expression evaluation (Table 1)
  $$\sigma \vdash (\ldots, \epsilon_1, \ldots) \xrightarrow{\alpha} (\ldots, \epsilon_1', \ldots)$$
  Using that ϵ_1 is a subterm of ϵ_0, by the inductive hypothesis, $g(\sigma) \vdash g(\epsilon_1) \xrightarrow{g(\alpha)} g(\epsilon_1')$. Since $g(\ldots, \eta, \ldots) = (\ldots, g(\eta), \ldots)$, we conclude that
  $$g(\sigma) \vdash g(\ldots, \epsilon_1, \ldots) \xrightarrow{g(\alpha)} g(\ldots, \epsilon_1', \ldots)$$
- Case $\epsilon = (\ldots, w, \ldots)$ with $[\![w]\!] = \mathbf{xcpt}$:
  According to the semantics in Table 1, $\epsilon' = w$ and $\alpha = \tau$. Since w is a subterm of ϵ_0, by (16.2), $[\![g(w)]\!] = \mathbf{xcpt}$. We can then infer that
  $$g(\sigma) \vdash (\ldots, g(w), \ldots) \xrightarrow{\alpha} g(w)$$
- Case $\epsilon = f\ \epsilon_1$ and $\sigma \vdash \epsilon_1 \xrightarrow{\alpha} \epsilon_1'$:
  Since ϵ_1 is a subterm of ϵ_0, by the inductive hypothesis, $g(\sigma) \vdash g(\epsilon_1) \xrightarrow{g(\alpha)} g(\epsilon_1')$. It is then immediate that
  $$g(\sigma) \vdash g(f\ \epsilon_1) \xrightarrow{g(\alpha)} g(f\ \epsilon_1')$$
- Case $\epsilon = \mathit{pf}\, w$ with $\mathit{pf}([\![w]\!]) = v$:
  From Table 1, $\epsilon' = v : \mathit{pf}\, w$ and $\alpha = \tau$. Therefore $g(\epsilon) = \mathit{pf}\, g(w)$ and $g(\alpha) = \tau$. If $g(v : \mathit{pf}\, w) = v' : \mathit{pf}\, g(w)$, then $\mathit{pf}([\![g(w)]\!]) = v'$ (by (8.4)), so that
  $$g(\sigma) \vdash \mathit{pf}\, g(w) \xrightarrow{g(\alpha)} g(v : \mathit{pf}\, w)$$
- Case $\epsilon = h\, w$:
  From Table 1, $\epsilon' = v : h\, w$ and $\alpha = v \lhd h\, w$. Therefore $g(\epsilon) = h\, g(w)$. Let $g(v : h\, w) = v' : h\, g(w)$. Then $g(\epsilon') = v' : h\, g(w)$ and
  $$g(\sigma) \vdash g(h\, w) \xrightarrow{v' \lhd h\, g(w)} g(v : h\, w) \qquad \square$$

Lemma 17 *Suppose that $c \in \mathit{Com}$ is stable w.r.t. dependency specification P. Then, $\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle$ iff $\langle g(c), g(\sigma), g(s)\rangle \xrightarrow{g(\alpha)} \langle g(c'), g(\sigma'), g(s')\rangle$.*

PROOF. Note first that, because of Lemmas A.3 and 11, only one implication needs to be shown. The proof proceeds by induction on the structure of the derivation of $\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle$.

- Case $c = \mathbf{skip}$:
  In this case, $c' = *$, $\sigma = \sigma'$, $s = s'$ and $\alpha = \tau$. Then

$$\langle g(c) = \mathbf{skip},\ g(\sigma),\ g(s)\rangle \xrightarrow{\tau = g(\alpha)} \langle g(c'),\ g(\sigma'),\ g(s')\rangle$$

- Case $c = x := \beta$:
  Here, $c' = *$, $\sigma' = \sigma[\beta/x]$, $s' = s$, and $\alpha = \tau$. Then

$$\begin{tikzcd}
\langle g(c),\ g(\sigma),\ g(s)\rangle \arrow[d, phantom, "\|"] & \\
\langle x := g(\beta),\ g(\sigma),\ g(s)\rangle \arrow[r, "\tau = g(\alpha)"] & \langle *,\ g(\sigma)[g(\beta)/x],\ g(s)\rangle \arrow[d, phantom, "\|"] \\
 & \langle g(c'),\ g(\sigma'),\ g(s')\rangle
\end{tikzcd}$$

- Cases $c = c_0; c_1$ (where $c_0 \neq *$) and $c = \mathbf{try}\ c_0\ \mathbf{catch}\ c_1$ (where $c_0 \neq \mathbf{throw}$):
  Note that $c_0 \preccurlyeq c$ and therefore, by Rem. A.2, c_0 is stable.
  The derivation of $\langle c,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c',\ \sigma',\ s'\rangle$ has respectively the forms:

$$\frac{\langle c_0,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0',\ \sigma',\ s'\rangle}{\langle c,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0'; c_1,\ \sigma',\ s'\rangle} \qquad \frac{\langle c_0,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle c_0',\ \sigma',\ s'\rangle}{\langle c,\ \sigma,\ s\rangle \xrightarrow{\alpha} \langle \mathbf{try}\ c_0'\ \mathbf{catch}\ c_1,\ \sigma',\ s'\rangle}$$

  By the inductive hypothesis, $\langle g(c_0),\ g(\sigma),\ g(s)\rangle \xrightarrow{\alpha} \langle g(c_0'),\ g(\sigma'),\ g(s')\rangle$. The result follows from $g(c_0; c_1) = g(c_0); g(c_1)$, and $g(\mathbf{try}\ c_0\ \mathbf{catch}\ c_1) = \mathbf{try}\ g(c_0)\ \mathbf{catch}\ g(c_1)$.
- Case $c = \mathbf{if}\ \beta\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$:
  We only consider the case when $[\![\beta]\!] = \mathbf{true}$. The case when $[\![\beta]\!] \neq \mathbf{true}$ is completely analogous.
  From the only semantics rule applicable at this instance, $\alpha = \tau$, $c' = c_0$, $\sigma' = \sigma$, and $s' = s$. Moreover, $g(c) = \mathbf{if}\ g(\beta)\ \mathbf{then}\ g(c_0)\ \mathbf{else}\ g(c_1)$. Since c is stable, by (16.1), $[\![g(\beta)]\!] = \mathbf{true}$. Finally, using the semantics rule for the **if** command,

$$\langle \mathbf{if}\ g(\beta)\ \mathbf{then}\ g(c_0)\ \mathbf{else}\ g(c_1),\ g(\sigma),\ g(s)\rangle \xrightarrow{\tau} \langle g(c_0),\ g(\sigma),\ g(s)\rangle$$

- Case $c = \mathbf{while}\ \epsilon\ \mathbf{do}\ c_0\ \mathbf{end}$:
  In this case, $c' = \mathbf{if}\ \epsilon\ \mathbf{then}\ c_0; c\ \mathbf{else\ skip}$, $\alpha = \tau$, $\sigma' = \sigma$ and $s' = s$. Since $g(c) = \mathbf{while}\ g(\epsilon)\ \mathbf{do}\ g(c_0)\ \mathbf{end}$ and $g(c') = \mathbf{if}\ g(\epsilon)\ \mathbf{then}\ g(c_0);g(c)\ \mathbf{else\ skip}$, the result follows trivially.

- Case $c = \textbf{try throw catch } c_1$:
  Note that $\alpha = \tau$, $c' = c_1$, $\sigma' = \sigma$, $s' = s$. The result follows trivially from $g(c) = \textbf{try throw catch } g(c_1)$.
- Case $c = r[\epsilon]$ and $\sigma \vdash \epsilon \xrightarrow{\alpha} \epsilon'$:
  Here, $c' = r[\epsilon']$, $\sigma' = \sigma$, and $s' = \begin{cases} s & \text{if } \alpha = \tau \\ s[v/h\,w] & \text{if } \alpha = (v \lhd h\,w) \end{cases}$

  By Lemma A.4, $g(\sigma) \vdash g(\epsilon) \xrightarrow{g(\alpha)} g(\epsilon')$. We can then derive

  $$\langle g(r)[g(\epsilon)],\ g(\sigma),\ g(s)\rangle \xrightarrow{g(\alpha)} \langle g(r)[g(\epsilon')],\ g(\sigma),\ s''\rangle$$

  where we have extended the permutation of commands over reduction contexts (so that $g(r)[\cdot]$ is like $r[\cdot]$ only that annotated values have been permuted according to g) and

  $$s'' = \begin{cases} g(s) & \text{if } g(\alpha) = \tau \\ g(s)[v'/h\,w'] & \text{if } g(\alpha) = (v' \lhd h\,w') \end{cases}$$

  $$= \begin{cases} g(s') & \text{if } \alpha = \tau \\ g(s[v/h\,w]) & \text{if } \alpha = (v \lhd h\,w) \quad \text{(by Lemma 9)} \end{cases}$$

  $$= g(s')$$

  Notice finally that $g(r)[g(\epsilon')] = g(r[\epsilon']) = g(c')$, and $g(\sigma) = g(\sigma')$.
- Case $c = r[w]$, where $[\![w]\!] = \textbf{xcpt}$:
  We know that $\langle r[w],\ \sigma,\ s\rangle \xrightarrow{\tau} \langle \textbf{throw},\ \sigma,\ s\rangle$. Since c is stable, $[\![g\,w]\!] = \textbf{xcpt}$ (16.2). This and $g(r[w]) = g(r)[g(w)]$ imply that

  $$\langle g(r[w]),\ g(\sigma),\ g(s)\rangle \xrightarrow{\tau} \langle \textbf{throw},\ g(\sigma),\ g(s)\rangle$$

  □

Theorem 19 *If $c \in Com$ 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 $\sim$).*

PROOF. Let g be an arbitrary secret permuter for P, and define:

$$R \triangleq \{\ (\langle c_1[I],\ \sigma_1,\ s_1\rangle, \langle g(c_1)[g^{-1}],\ g(\sigma_1),\ g(s_1)\rangle)\ \mid$$

$$\langle c,\ \sigma,\ s\rangle \Rightarrow \langle c_1,\ \sigma_1,\ s_1\rangle\ \}$$

Suppose that $\langle c_1[I],\ \sigma_1,\ s_1\rangle\ R\ \langle g(c_1)[g^{-1}],\ g(\sigma_1),\ g(s_1)\rangle$, and suppose first that

$$\langle c_1[I],\ \sigma_1,\ s_1\rangle \xrightarrow{\gamma} \langle c',\ \sigma_2,\ s_2\rangle$$

From the semantic definition of the relabelling operator (7), there must exist α and c_2 such that $\gamma = [\![I(s_1,\alpha)]\!] = [\![\alpha]\!]$, $c' = c_2[I]$ and

$$\langle c_1,\ \sigma_1,\ s_1\rangle \xrightarrow{\alpha} \langle c_2,\ \sigma_2,\ s_2\rangle \tag{A.2}$$

Since c_1 is a stable command (because $\langle c,\ \sigma,\ s\rangle$ is stable and c_1 is reachable from $\langle c,\ \sigma,\ s\rangle$ by definition of R) Lemma 17 can be applied to equation (A.2) to yield:

$$\langle g(c_1),\ g(\sigma_1),\ g(s_1)\rangle \xrightarrow{g(\alpha)} \langle g(c_2),\ g(\sigma_2),\ g(s_2)\rangle \tag{A.3}$$

Since $P, s_1 \vdash \alpha\ ok$ (by Flow Compatibility), $g^{-1}(g(s_1), g(\alpha)) = \alpha$ (Lemma A.1). From equation (7),

$$\langle g(c_1)[g^{-1}],\ g(\sigma_1),\ g(s_1)\rangle \xrightarrow{\gamma} \langle g(c_2)[g^{-1}],\ g(\sigma_2),\ g(s_2)\rangle$$

and $\langle c_2[I],\ \sigma_2,\ s_2\rangle\ R\ \langle g(c_2)[g^{-1}],\ g(\sigma_2),\ g(s_2)\rangle$.

Suppose now that $\langle g(c_1)[g^{-1}],\ g(\sigma_1),\ g(s_1)\rangle \xrightarrow{\gamma} \langle c',\ \sigma',\ s'\rangle$. There must exist c_2, σ_2, s_2 and α such that equation (A.3) holds, together with $\gamma = [\![g^{-1}(g(s_1), g(\alpha))]\!]$, $c' = g(c_2)[g^{-1}]$, $\sigma' = g(\sigma_2)$ and $s' = g(s_2)$. Equation (A.2) follows from Lemma 17, and the result is then obtained like in the previously analyzed case. □

Lemma A.5 *Consider a set $\{\xi_i\}_{i\in\mathcal{I}}$ satisfying conditions P1–P3 (p. 19). Then, for each configuration $\xi = \langle c,\ \sigma,\ s\rangle$ that is reachable from ξ_0, there is an $i \in \mathcal{I}$ and a secret permuter g such that $\xi = g(\xi_i)$.*

PROOF. By induction on the length of the shortest path from ξ_0 to ξ. If the path has length zero, take $i = 0$ and $g = I$. Otherwise, let ξ' be on the path such that $\xi' \xrightarrow{\alpha} \xi$. By the inductive hypothesis, there must exist $j \in \mathcal{I}$, $\xi_j = \langle c',\ \sigma',\ s'\rangle$ and g such that $g(\xi_j) = \xi'$. By P2, c' is a stable command.

Then $g(c')$ is also stable (Lemma A.3) and therefore $g^{-1}(\xi') \xrightarrow{g^{-1}(\alpha)} g^{-1}(\xi)$ (Lemma 17). Note that $g^{-1}(\xi') = g^{-1}(g(\xi_j)) = \xi_j$. Using P3, choose $i \in \mathcal{I}$ and permuter g' so that $g^{-1}(\xi) = g'(\xi_i)$. Then, $\xi = (g \circ g')(\xi_i)$, and the result follows from Prop. 12. □

Theorem 20 *Let P be a dependency specification and $\langle c_0, \sigma_0, s_0\rangle$ an initial configuration. If there is a set of configurations $\{\xi_i\}_{\mathcal{I}}$ satisfying P1–P3, then c_0 is admissible (for σ_0, s_0, P and $\sim$).*

PROOF. By Theorem 19, it suffices to show that ξ_0 is both stable and flow compatible. Assume $\xi_0 \Rightarrow^* \xi = \langle c, \sigma, s\rangle$. By Lemma A.5, $\xi = g(\xi_i)$ where $\xi_i = \langle c', \sigma', s'\rangle$ for some $i \in \mathcal{I}$.

- (Stability) By P2, c' is stable. Since $c = g(c')$, c is stable (Lemma A.3).
- (Flow compatibility) If $\xi \xrightarrow{\alpha} q$, then $\xi_i \xrightarrow{g^{-1}(\alpha)} g^{-1}(q)$ (Lemma 17). By P3, $P, s' \vdash g^{-1}(\alpha)\ ok$. That is, $P, s \vdash \alpha\ ok$ (Lemma 11). □

Lemma A.6 *For all c, σ, s, α, the sets*

$$\delta_\alpha(\langle c, \sigma, s\rangle) = \{\langle c', \sigma', s'\rangle \mid \langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle\}$$
$$Acts(\langle c, \sigma, s\rangle) = \{f\ w \mid \exists c', \sigma', s', v. \langle c, \sigma, s\rangle \xrightarrow{v \lhd f\ w} \langle c', \sigma', s'\rangle\}$$

are finite.

Lemma 21 *Suppose $\langle c_0,\ \sigma_0,\ s_0\rangle$ is stable and admissible for dependency specification P. Then for all behaviours*

$$\langle c_0,\ \sigma_0,\ s_0\rangle \Rightarrow \langle c_1,\ \sigma_1,\ s_1\rangle \xrightarrow{v \lhd f\, w} \langle c_1',\ \sigma_1',\ s_1'\rangle$$

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

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

is finite.

PROOF. Say

$$\langle c',\ \sigma',\ s'\rangle \overset{\Theta}{\Rightarrow} \langle c'',\ \sigma'',\ s''\rangle$$

iff

$$\langle c',\ \sigma',\ s'\rangle = \langle c'_0,\ \sigma'_0,\ s'_0\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \langle c'_n,\ \sigma'_n,\ s'_n\rangle = \langle c'',\ \sigma'',\ s''\rangle$$

and Θ is the sequence $\alpha_1, \ldots, \alpha_n$. Define $[\![\Theta]\!]$ as the sequence $[\![\alpha_1]\!], \ldots, [\![\alpha_n]\!]$, and $g(\Theta)$ as the sequence $g(\alpha_1), \ldots, g(\alpha_n)$. Let

$$\delta_\Theta(\langle c, \sigma, s\rangle) = \{\langle c', \sigma', s'\rangle \mid \langle c, \sigma, s\rangle \overset{\Theta}{\Rightarrow} \langle c', \sigma', s'\rangle\}$$

and it follows from Lemma A.6 that $\delta_\Theta(\langle c, \sigma, s\rangle)$ and even $\delta_{[\![\Theta]\!]}(\langle c[I], \sigma, s\rangle)$ is finite.

Take any Θ of minimal length such that

$$\langle c_0,\ \sigma_0,\ s_0\rangle \overset{\Theta}{\Rightarrow} \langle c_1,\ \sigma_1,\ s_1\rangle \xrightarrow{\alpha} \langle c'_1,\ \sigma'_1,\ s'_1\rangle \tag{A.4}$$

and $P, s_1 \not\vdash \alpha\ ok$. Notice that, by requiring the length of Θ to be minimal, it is guaranteed that every action in Θ is admissible at its corresponding context in (A.4).

Let now g be any secret permuter. Given that $\langle c_0,\ \sigma_0,\ s_0\rangle$ is stable, it follows from Lemma 17 that

$$\begin{aligned} \langle g(c_0),\ g(\sigma_0),\ g(s_0)\rangle \ &\overset{g(\Theta)}{\Rightarrow}\ \langle g(c_1),\ g(\sigma_1),\ g(s_1)\rangle \\ &\xrightarrow{g(\alpha)} \langle g(c'_1),\ g(\sigma'_1),\ g(s'_1)\rangle \end{aligned} \tag{A.5}$$

Then, by Lemma 11, every action in $g(\Theta)$ is admissible at its corresponding context in (A.5) and $P, g(s_1) \not\vdash g(\alpha)\ ok$. By the definition of $[g^{-1}]$ and Corollary A.1,

$$\begin{aligned} \langle g(c_0)[g^{-1}],\ g(\sigma_0),\ g(s_0)\rangle \ &\overset{[\![\Theta]\!]}{\Rightarrow}\ \langle g(c_1)[g^{-1}], g(\sigma_1), g(s_1)\rangle \\ &\xrightarrow{[\![g(\alpha)]\!]} \langle g(c'_1)[g^{-1}],\ g(\sigma'_1),\ g(s'_1)\rangle \end{aligned}$$

By admissibility,

$$\langle c_0[I],\ \sigma_0,\ s_0\rangle \overset{[\![\Theta]\!]}{\Rightarrow} \langle c_2[I],\ \sigma_2,\ s_2\rangle \xrightarrow{[\![g(\alpha)]\!]} \langle c'_2[I],\ \sigma'_2,\ s'_2\rangle \tag{A.6}$$

W.l.o.g. assume $\alpha = v \lhd f\ w$. Then $[\![g(\alpha)]\!] = [\![v' \lhd f\ g(w)]\!] = v' \lhd f\ [\![g(w)]\!]$ for some value v', and therefore (A.6) shows that, for each secret permuter g,

$$f\,[\![g(w)]\!] \in \bigcup_{\langle c_2[I],\ \sigma_2,\ s_2\rangle \in \delta_{[\![\Theta]\!]}(\langle c_0[I],\ \sigma_0,\ s_0\rangle)} \mathit{Acts}(\langle c_2[I],\ \sigma_2,\ s_2\rangle)$$

The set on the right is a finite union of finite sets, by Lemma A.6, and is therefore finite. Therefore, there is only a finite number of distinct $[\![g(w)]\!]$. $\square$

References

[1] M. Dam, P. Giambiagi, Confidentiality for mobile code: The case of a simple payment protocol, in: Proceedings of 13th IEEE Computer Security Foundations Workshop, IEEE, Cambridge, England, 2000, pp. 233–244.

[2] A. Sabelfeld, A. C. Myers, Language-Based Information-Flow Security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5–19.

[3] M. Abadi, A. Benerjee, N. Heintze, J. G. Riecke, A core calculus of dependency, in: Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, San Antonio, TX, 1999, pp. 147–160.

[4] E. S. Cohen, Information transmission in sequential programs, in: R. A. DeMillo, D. P. Dobkin, A. K. Jones, R. J. Lipton (Eds.), Foundations of Secure Computation, Academic Press, 1978, pp. 297–335.

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

[6] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security 4 (3) (1996) 167–187.

[7] A. Sabelfeld, D. Sands, A per model of secure information flow in sequential programs, Higher Order and Symbolic Computation 14 (1) (2001) 59–91.

[8] D. Volpano, Secure introduction of one-way functions, in: Proceedings of 13th IEEE Computer Security Foundations Workshop, IEEE, Cambridge, England, 2000, pp. 246–254.

[9] M. Abadi, A. D. Gordon, A bisimulation method for cryptographic protocols, Nordic Journal of Computing 5 (4) (1998) 267–303.

[10] J. C. Mitchell, Foundations for Programming Languages, MIT Press, 1996.

[11] A. Sabelfeld, D. Sands, A PER model of secure information flow in sequential programs, in: Proceedings of the 8th European Symposium on Programming, Vol. 1576 of LNCS, Springer, Amsterdam, 1999, pp. 40–58.

[12] D. Naumann, Soundness of data refinement for a higher order imperative language, Theoretical Computer Science 278 (2002) 271–301.

[13] P. Laud, Handling encryption in an analysis for secure information flow, in: Proceedings of Programming Languages and Systems, 12th European Symposium On Programming, ESOP 2003, no. 2618 in LNCS, Springer, 2003, pp. 159–173.

[14] A. W. Roscoe, M. H. Goldsmith, What is intransitive noninterference?, in: Proceedings of 12th IEEE Computer Security Foundations Workshop, IEEE, Mordano, Italy, 1999, pp. 228–238.

[15] S. Zdancewic, A. Myers, Robust declassification, in: Proceedings of 14th IEEE Computer Security Foundations Workshop, IEEE, Nova Scotia, Canada, 2001, pp. 15–23.