Type-Based Enforcement of Non-Interference for Choreographic Programming

Marco Bertoni[1], Saverio Giallorenzo[2,3] and Marco Peressotti[4]

[1]*Inria Paris, France*
[2]*Università di Bologna, Italy*
[3]*Inria Sophia Antipolis, France*
[4]*University of Southern Denmark, Denmark*

Abstract

Choreographies describe distributed protocols from a global viewpoint, enabling correct-by-construction synthesis of local behaviours. We develop a policy-parametric type system that prevents information leaks from $high$-security data to low-security observers, handling both explicit and implicit flows through a program-counter discipline. The system supports recursive procedures via a procedure context that we reconstruct through constraint generation. We prove termination-insensitive non-interference with respect to a standard small-step semantics.

Keywords

Choreographic programming, Non-interference, Security

1. Introduction

Distributed software implements protocols through communicating components – authenticating users, reconciling payments, etc.. Two paramount concerns in distributed software are *correct coordination* among participants and *confidentiality* of handled data. Choreographic programming languages [1] address coordination by describing protocols from a global viewpoint, from which one can derive compliant distributed software. This work tackles the second concern: *ensuring that choreographies do not leak secret information to public observers.*

The challenges of ensuring confidentiality emerge in realistic protocols where control flow depends on sensitive data. We exemplify the concept with a simple password-recovery scenario. A requester `r` submits an email address to a service `s`, which may then task a mailer service `m` to send a reset link. While the protocol concerns a few straightforward steps, even seemingly innocuous design choices can inadvertently reveal sensitive information. Whether the email corresponds to a registered account is sensitive information: revealing it enables username enumeration attacks. Thus, we must treat the outcome of a check `exists(email)` by `s` as secret, while the behaviour observable by `r` is considered public.

Let us visualise the issue with an insecure choreography for that scenario. In the choreography, `a.e -> b.x` means that `a` sends the evaluation of expression `e` to `b`, which stores the received value into its local variable `x`.

Insecure choreography example

```
r.email -> s.email
if s.exists(email) then
  s.email -> m.email; s."email sent" -> r.msg
else s."unknown user" -> r.msg
```

Although the boolean returned by `exists(email)` is never sent directly to `r`, the requester can distinguish executions by observing different response messages. As a consequence, `r` learns the sensitive bit of information: whether the account exists. The leak arises not from an explicit transmission of secret data, but from a control-flow dependency that produces observable differences.

ICTCS 2026: Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy
0009-0000-9224-6005 (M. Bertoni); 0000-0002-3658-6395 (S. Giallorenzo); 0000-0002-0243-0480 (M. Peressotti)

We can remove the leak by ensuring that branching on `exists(email)` affects only internal communication with the mailer, while keeping the observable behaviour at `r` uniform.

Secure choreography variant

```
r.email -> s.email
if s.exists(email) then s.email -> m.email else skip
s."check your inbox" -> r.msg
```

In the secure variant, executions that differ only in the secret predicate produce identical observations at `r`. This example illustrates *implicit flow* of information, where secrets influence which actions occur, in contrast to *explicit flow*, where secret values are directly assigned or transmitted to public locations.

To detect and prevent such leaks systematically, we require a formal account of how information may flow (both explicitly and implicitly) in a choreography. Following Denning [2], we assign each datum a *security class* drawn from a lattice, whose order relation $\sqsubseteq$ models relative sensitivity. Together with the class assignment, the lattice structure defines a *flow policy*, specifying when information is allowed to move between locations of different sensitivity: data labelled ℓ_1 may flow to contexts labelled ℓ_2 only if $\ell_1 \sqsubseteq \ell_2$.

In this work, we focus on *non-interference* [3], which requires that variations in higher-security inputs must not influence lower-security outputs [4]. Intuitively, secure data should not affect what an external observer can learn from a program's execution, i.e., interfere with the observed behaviour – precisely the violation exhibited by our first attempt at implementing password recovery.

Unlike in sequential programs, information flow in choreographies is determined not only by local computations but also by the global communication structure prescribed by the protocol. A confidential decision taken by one participant may influence which communications occur and, consequently, the behaviour observed by other participants, even when the secret itself is never transmitted. Because choreographies explicitly describe both communication and control flow, they provide a natural foundation for static reasoning about such protocol-level information flows and provide several advantages

- First, all communication between processes appears explicitly in the choreography, making potential information-flow channels syntactically visible – in our example, the message from `s` to `r` immediately identifies a possible leak.
- Second, security is verified for the entire protocol through a single global judgement, rather than by composing separate guarantees for individual participants.
- Third, choreographies capture both local control flow and inter-process coordination, allowing us to detect implicit flows that arise when a choice in one participant influences the observable behaviour of others.
- Finally, once a choreography is shown to satisfy non-interference, endpoint projection preserves this property by construction [5]. Thus, we obtain end-to-end guarantees, from a verified global specification to the generated distributed implementation.

This combination of global reasoning, explicit communication structure, and correctness-preserving compilation makes choreographic programming a natural foundation for enforcing non-interference in distributed systems.

Our approach. We develop a *type-based enforcement* of non-interference for the textbook choreographic programming language Recursive Choreographies [1]. Our type system is *policy-parametric*: given a user-specified flow policy over a lattice $\mathscr{L}$ and an observation level low, the type system rejects choreographies where high-security data could influence low-observable behaviour.

Our type system centres on a judgement of the form $\Delta; \Gamma; pc \vdash C$, where Δ specifies how procedures may be safely invoked at different control levels, Γ assigns security classes to variables, and the *program counter* $pc \in \mathscr{L}$ tracks the current control level (providing the context for capturing implicit flow). This definition generalises standard information-flow type systems to the choreographic setting, where information flows both through message-passing between processes and through control dependencies in the global protocol.

We rely on two fundamental technical devices to obtain our results, embodied by pc and Δ. First, we adopt a standard *program-counter discipline*: the control level pc records the influence of secret data on the current execution point. When branching on a secret (like `exists(email)` in our example), pc is raised, and subsequent actions are checked in this elevated context. This mechanism prevents low-observable behaviour from depending on high-security control flow, ruling out the implicit flow in our first password-recovery example. Second, to support modular verification in the presence of recursion, we introduce a *procedure context* Δ that assigns to each procedure and control level a set of admissible flow policies. Intuitively, Δ serves as a contract: $\Delta; \Gamma; pc \vdash X(\bar{p})$ ensures that calling the procedure $X(\bar{p})$ under Γ at control level pc preserves non-interference. We develop a constraint-based reconstruction procedure that computes Δ from the procedure definitions themselves, thereby automating this verification step.

Our enforcement target is a standard small-step semantics for choreographies. Following established practice in language-based information-flow security [4], we adopt a *termination-insensitive* attacker model, in which divergence is not considered observable. While stronger notions exist, this choice strikes a pragmatic balance between expressiveness (it allows programmes whose runtime depends on secrets) and security guarantees.

Our main result (Theorem 1) establishes the soundness of the type system with respect to termination-insensitive non-interference.

Structure of the Paper. In Section 2, we introduce the reference choreographic language and its small-step operational semantics. In Section 3, we present a policy-parametric information-flow type system for the reference choreographic language, tracking both explicit and implicit flow and supporting recursive procedures given a valid procedure context Δ. In Section 4 we define and prove a soundness theorem, establishing that well-typed choreographies satisfy termination-insensitive non-interference w.r.t. the language semantics. In Section 5, we define and prove the correctness of a constraint-based reconstruction algorithm that, given a collection of procedure definitions, computes a context Δ under which these procedures are well-typed – the expected correctness criteria. We discuss related work and concluding remarks in, respectively, Sections 6 and 7. Full proofs are available in [6].

2. Preliminaries on Choreographies

We recall essential notions of choreographic programming languages [1] and introduce the language intended for users to write distributed programs, called *Recursive Choreographies* (RC). Choreographies formally describe intended collaborative behaviour of processes in concurrent and distributed systems, acting as protocols specifying how components interact to achieve shared goals. Although choreographies are written is a way that reminds traditional local programs, they execute through transitions representing distributed communication and computation steps. A choreography represents the global state of all participants and encompasses the actions that collectively implement the protocol.

2.1. Syntax

Processes are participants in a choreography, performing local computation and communicating with others. We range over process names by $p, q, r, s \in \mathsf{PName}$. Choreographies in RC are given by the

following grammar:

$\mathscr{C}$	::=	$\{X_i(\bar{p}_i) = C_i\}_{i \in I}$	choreographic procedure definitions
C	::=	$I; C$	instruction sequencing
	|	$\mathbf{0}$	terminated choreography
I	::=	$p.e \to q.x$	p sends the result of e to q which assigns it to x
	|	$p \to q[L]$	p sends the constant L to q
	|	$p.x := e$	p assigns to x the result of e
	|	if $p.e$ then C_1 else C_2	p evaluates e and chooses to continue as C_1 or C_2
	|	$X(\bar{p})$	call procedure X with actual arguments $\bar{p}$
	|	$r : X(\bar{p}).C'$	(runtime term, r has yet to enter the call)
e	::=	$v \mid x \mid f(\bar{e})$	local values, variables, and function calls

Here $\mathscr{C}$ denotes a context of procedure definitions; C denotes a choreography – either terminated $\mathbf{0}$ or that composes in sequence instruction I and continuation C. I denotes instructions; from top to bottom, we find communication, selection, local assignment, conditional, and procedure call. e denotes local expressions (constants $v \in \mathsf{Val}$, variables $x \in \mathsf{Var}$, function calls $f(\bar{e})$). The term $r : X(\bar{p}).C'$ is not part of the front facing syntax of the language and is only used in the semantics to model that processes enter a procedure call independently, without any synchronisation as we will discuss in the next section. We omit trailing $\mathbf{0}$ when confusion is unlikely.

2.2. Semantics

We define the semantics of Recursive Choreographies using a small-step operational semantics [7], forming a labelled transition system. Configurations have form $\langle C, \Sigma, \mathscr{C} \rangle$ where C is the choreography, Σ the *choreographic store*, and $\mathscr{C}$ the procedure context.

A *process store* $\sigma : \mathsf{Var} \to \mathsf{Val}$ models one process's memory. We call PStore for the set of all process stores. A *choreographic store* $\Sigma : \mathsf{PName} \to \mathsf{PStore}$ models the entire system's memory. We write $\Sigma[p.x \mapsto v]$ for updating store Σ so local variable x of process p maps to v. We call CStore the set of all choreographic stores.

Given process store σ, expression e, and value v, we write $\sigma \vdash e \downarrow v$ when e evaluates to v under σ. We do not specify a system for deriving propositions of the kind $\vdash f(\bar{v}) \downarrow v$, since it is not important for our development: this system would depend on how functions are defined, which we choose to abstract from. Instead, we just assume that such a system exists, and that for any f and $\bar{v}$, it is always possible to derive $\vdash f(\bar{v}) \downarrow v$ for some $\bar{v}$.

Transition labels record the kind of step performed – internal at p ($\tau@p$), communication ($p.v \to q$), selection ($p \to q[L]$), or the outcome of a conditional at p (p.then/p.else).

We report the rules of the semantics of Recursive Choreographies in Figure 1. Rule COM models the communication of a value computed at runtime: p evaluates the expression e against its local store and sends the result v to q which stores it under variable x in its local store. Similarly, Rule SEL models the communication of a constant L, called a selection.[1] Rules COND-THEN, COND-ELSE model conditional branching: process p evaluates the guard e locally and the choreography continues as the corresponding branch – the operator $\fatsemi$ implements sequential composition ($C; C'$ is not part of the grammar) by grafting the continuation C in place of any $\mathbf{0}$ in C_1 and C_2. Rules CALL-FIRST and CALL-ENTER model decentralised calls to a choreographic procedure i.e., without synchronising the processes that enter the procedure. These rules rely on the runtime term $r : X(\bar{p}).C'$ to track that process r has not entered the call and thus cannot perform any action from its continuation via the delay rules. Rule CALL-FIRST unfolds the procedure definition from the context $\mathscr{C}$ and instantiates formal process parameters $\bar{q}$ with the actual ones $\bar{p}$ in its body C. The selection of which process enters the call first and which will do it later via CALL-ENTER is nondeterministic since the premise $\{p_0, \ldots, p_n\} = \{r_0, r_1, \ldots, r_n\}$ allows to

[1] The construct of selections plays a critical role in the handling of branching constructs in endpoint projection which is beyond the scope of this work. We refer the curious reader to [1, Chapter 6].

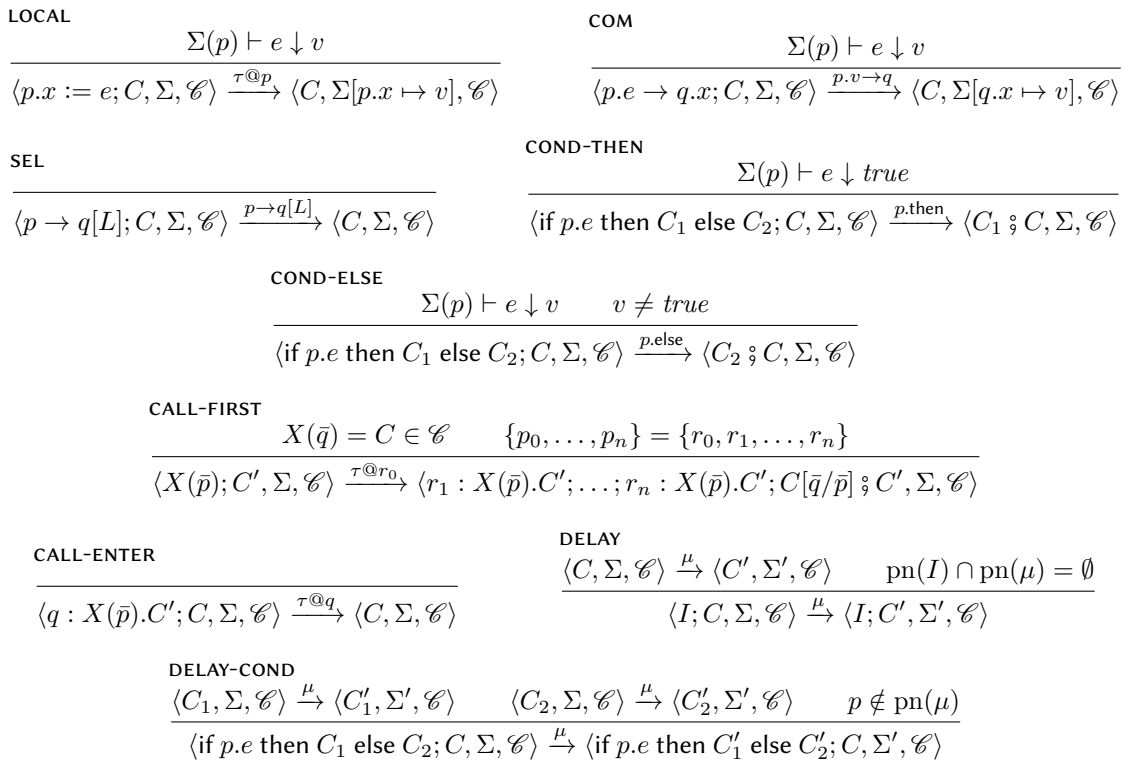

Figure 1: Operational Semantics.

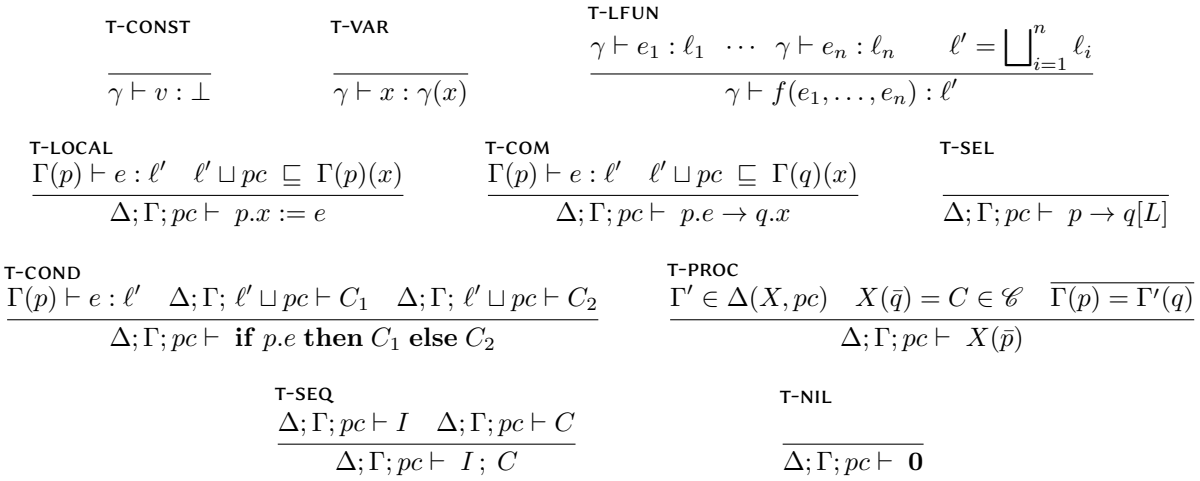

Figure 2: Typing rules.

arbitrarily reindex $\bar{p}$. Rules DELAY and DELAY-COND model that actions at independent processes are executed concurrently by delaying the execution of an instruction or branch selection. The rules use the operator pn which returns the set of process names involved in an instruction I or label μ.

3. Type System

Our goal is to statically verify that a choreography complies with a user-specified information-flow policy. Following the classical approach by Volpano et al. [4], we enforce non-interference through

a syntax-directed type system inspired by non-interference standard techniques [8, 9], adapted to choreographies. Intuitively, the type system ensures that information flows only from less confidential data towards equally or more confidential data. In the choreographic setting, this discipline must account not only for assignments and communications, but also for control flow and recursive procedure calls.

We model Denning's security classes [2] using *Security labels*, elements of a chosen complete lattice $(\mathscr{L}, \sqsubseteq)$ where $\bot$ denotes its bottom element. A security environment assigns a security label to every variable. Formally, a *process security labelling* $\gamma : \mathsf{Var} \to \mathscr{L}$ assigns security labels to variables accessed by a process and a *choreographic security labelling* $\Gamma : \mathsf{PName} \to (\mathsf{Var} \to \mathscr{L})$ maps process names to their respective labellings.

The type system is built around three ideas.

First, whenever an expression is assigned to a variable or transmitted to another participant, the recipient must be at least as confidential as the information being transferred. Expressions therefore receive the least upper bound of the labels of the variables they reference. Consequently, an assignment such as $p.x := y + z$ is accepted only if the security level of $p.x$ dominates the join of the labels of $p.y$ and $p.z$. Communications are treated analogously: sending a value from p to q is equivalent to assigning that value to a variable owned by q. This reflects our assumption that communication channels are private, so only the sender and receiver observe the transmitted value.

Second, explicit-flow checks alone are insufficient since confidential information may also influence computation through control flow. Consider the following choreography.

$$\text{if } p.(a == 0) \text{ then } p.x := 0 \text{ else } p.x := y + z \tag{1}$$

To account for such implicit flows, typing judgements are parametrised by a *program-counter label* pc, representing the security level of the current control context. Whenever a conditional is encountered, the label of its guard is joined with the current pc, and both branches are checked under this updated control level. As a result, every assignment or communication performed under confidential control is itself regarded as confidential.

Third, recursive procedures introduce an additional challenge as they may be called under contexts with different security levels and environments, in this sense, they can be regarded as polymorphic. Consider a simple procedure like $X(p, q) = p.x \to q.x$ and suppose that there is no explicit secret so we can label both variables with a low level. While calling the procedure in a context with a similarly low level is fine, it leaks information when called in a context with a higher level as shown in the following snippet.

$$X(p, q); \text{if } p.secret \text{ then } X(p, q) \text{ else } \mathbf{0} \tag{2}$$

We track this information through a *procedure security context* Δ that maps a procedure name X and program counter label pc to a set of choreographic security labellings ($\Delta(X, pc)$) under which the body of X is well typed when executed under pc. Section 5 presents an algorithm that reconstructs Δ from procedure declarations.

These three ingredients – security labels, the program-counter label, and the procedure context – are reflected directly in the typing judgements shown in Figure 2.

We define three mutually recursive typing judgements: the judgment $\gamma \vdash e : \ell$ captures that the expression e has sensitivity level $\ell \in \mathscr{L}$ under a security context characterised by the process security labelling γ, and the judgments $\Delta; \Gamma; pc \vdash I$ and $\Delta; \Gamma; pc \vdash C$ characterise that the instruction I and choreography C are well-typed under in the security context described by Δ, Γ, and pc.

Rules T-CONST, T-VAR, T-LFUN type expressions. Functions take the join of the labels of the arguments, thus assuming that local functions preserve labels and do not introduce extra leaks.

Rules T-LOCAL and T-COM type the two constructs that *write* into a store location, respectively a local assignment and a communication. These are the only rules that enforce an actual security condition: intuitively, the target must be at least as confidential as both the data being written and the control context in which the write happens. Note that rule T-COM treats communication as assignment between processes, thus assuming communication channels are private. Rule T-SEL types selections: since it only sends a fixed label L it introduces no explicit flow and any implicit flow is already captured by the

current pc. Rule T-COND accounts for implicit flows, it types a conditional by first typing the guard and then typing both branches under an updated program-counter label that tracks the sensitivity of the guard. In other words, T-COND is the only rule that can raise pc; all other rules preserve it. Rule T-PROC types a call to procedure X by checking that the context Δ allows a choreographic security labelling Γ' for X and the required pc that agrees on the labelling Γ of the calling context. Specifically, the third premise checks that $\Gamma(p_i) = \Gamma'(q_i)$ for any formal parameter q_i and corresponding actual parameter p_i – for conciseness we omit the set of procedure definitions $\mathcal{C}$ from the notation nor replicate the information about formal parameters in Δ. Rules T-NIL and T-SEQ are purely structural: $\mathbf{0}$ is always typable and sequencing just composes typable choreographies.

4. Soundness

We now proceed to prove the soundness of our type system w.r.t. termination-insensitive non-interference. To specify the theorem, we introduce some notation.

Natural Semantics. We introduce a natural semantics [10]. We define relation

$$\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^{M} \Sigma'$$

where M is a sequence of transition labels, defined as the smallest relation:

$$\langle \mathbf{0}, \Sigma, \mathscr{C} \rangle \Downarrow^{[]} \Sigma \qquad \frac{\langle C, \Sigma, \mathscr{C} \rangle \xrightarrow{\mu} \langle C', \Sigma', \mathscr{C} \rangle \qquad \langle C', \Sigma', \mathscr{C} \rangle \Downarrow^{M} \Sigma''}{\langle C, \Sigma, \mathscr{C} \rangle \Downarrow^{\mu::M} \Sigma''}$$

Attacker Observation Model. We formalise low-level observation by defining a low-equivalence relation $\Sigma_1 \equiv^{\Gamma}_{low} \Sigma_2$, which holds when the two stores agree on all variables whose labels in Γ are $\sqsubseteq low$. The only low-observable outputs are the values of such low-security variables. The attacker is thus able to observe only the *final state* of the execution, and not the labels trace produced by it.

Then, we state key assumptions needed for soundness:

Well-formed Procedure Context. For every definition $X(\bar{p}) = C \in \mathscr{C}$:

$$\mathrm{pn}(C) \subseteq \{\bar{p}\} \tag{3}$$

Well-typed Security Procedure Context. For every $X(\bar{p}) = C \in \mathscr{C}$ and $pc \in \mathscr{L}$:

$$\Delta\, X\, pc = \mathcal{G} \implies \forall \Gamma \in \mathcal{G}.\, \Delta; \Gamma; pc \vdash C \tag{4}$$

where $\mathcal{G}$ is the set of admissible environments for C at program counter pc.

Deterministic Expressions. Given process store σ, expression e, and values v_1, v_2:

$$\sigma \vdash e \downarrow v_1 \Rightarrow \sigma \vdash e \downarrow v_2 \Rightarrow v_1 = v_2$$

4.1. Instrumented Choreographies

The proof of non-interference proceeds by induction on executions. A direct induction based on the standard semantics of Recursive Choreographies is, however, insufficient. After evaluating a conditional whose guard depends on confidential data, two executions starting from low-equivalent stores may legitimately continue with different pieces of code. Consequently, the induction hypothesis cannot be applied, even though the executions remain indistinguishable to a low observer. Consider the

choreography from (1) and denote it by C. Given Σ_1, Σ_2 such that $\Sigma_1\ p.a = 0, \Sigma_2\ p.a = 1$ then we have the following transitions:

$$\langle C, \Sigma_1, \mathscr{C}\rangle \xrightarrow{p.\text{then}} \langle p.x := 0;\ \mathbf{0}, \Sigma_1, \mathscr{C}\rangle \qquad \langle C, \Sigma_2, \mathscr{C}\rangle \xrightarrow{p.\text{else}} \langle p.x := y + z;\ \mathbf{0}, \Sigma_2, \mathscr{C}\rangle$$

where the derivatives $p.x := 0;\ \mathbf{0}$ and $p.x := y + z;\ \mathbf{0}$ are syntactically distinct and, although this difference is confined to code executing under a high program counter, this is enough to preventing us from invoking the induction hypothesis.

To recover a usable inductive invariant, we instrument choreographies with explicit *brackets*, written $[C]$, marking fragments whose differences are not observable below the current security level – we call this extension of Recursive Choreographies *Instrumented Choreographies*. Intuitively, bracketed code records that two executions are allowed to diverge syntactically while remaining observationally equivalent for low observers.

Beyond brackets, there are two further differences w.r.t. Recursive Choreographies:

- *Flattening of instructions and choreographies:* Brackets may enclose sequences of instructions of arbitrary length (including empty ones), making terms such as $[\mathbf{0}]$ and $\mathbf{0}$ distinct.
- *Removal of the runtime term and label selection instructions:* Although not essential, this simplification eases the soundness proof. Its rationale is that both instructions have no impact on the final computed stores and are only needed in the semantics to support projection [1], which is omitted here.

This extended syntax supports a weaker notion of equivalence than syntactic equality. We write $C_1 \approx_{low} C_2$ for *low-equivalence* between choreographies: the relation compares choreographies structurally while ignoring the contents of bracketed subterms, equating them regardless of their internal structure.

Instrumented stores. We define $[\mathsf{CStore}]$, an extension of CStore containing possibly bracketed values $[\mathsf{Val}]$. Formally:

$$[\mathsf{Val}] ::= v \mid [v] \quad \text{with } v \in \mathsf{Val}$$

To relate instrumented and reference stores, we introduce two notions: *well-formedness* and *correctness*.

A $[\mathsf{CStore}]$ $[\Sigma]$ is *well formed* w.r.t. Γ when for all p, x, v:

$$[\Sigma]\, p.x = [v] \iff \Gamma\, p.x \not\sqsubseteq low$$

Where $\not\sqsubseteq$ encodes observability: $a \not\sqsubseteq low$ means low cannot see the value of a.

We define a *lowering* function $\lfloor[\Sigma]\rfloor$ that removes brackets from values while leaving unbracketed values unchanged. An instrumented store $[\Sigma]$ is *correct* with respect to a reference store Σ when $\lfloor[\Sigma]\rfloor = \Sigma$.

Low-equivalence of configurations. The notion of low-equivalence extends naturally to instrumented stores: two stores are low-equivalent when they coincide pointwise on all locations after ignoring bracketed values. We lift this notion to configurations: two configurations $\langle[C_1], [\Sigma_1], \mathscr{C}\rangle$ and $\langle[C_2], [\Sigma_2], \mathscr{C}\rangle$ are low-equivalent when $[C_1] \approx_{low} [C_2]$ and $[\Sigma_1] \approx_{low} [\Sigma_2]$.

Instrumented semantics. The instrumented semantics extends the reference semantics with explicit handling of bracketed values and choreographies and is parametric in the flow-policy Γ and the observation level low. Its purpose is twofold: to maintain well-formedness of instrumented stores throughout execution, and to record control-flow dependencies on high information by introducing brackets around branching continuations when needed.

Type System Extension. We add a rule to the type system so that a bracketed choreography can be typed at a lower program counter as long as its body executes at a strictly high security level, isolating high effects within brackets.

Relating Reference and Instrumented Executions. To relate the reference semantics to the instrumented one, it is helpful to think of the latter as a *decorated replay* of an ordinary execution. A *lifting* of a reference execution is then an instrumented execution that follows the same computational choices, but additionally records where high data/control may have influenced the run by introducing brackets around the corresponding continuations and values.

4.2. Main result

Theorem 1 (Termination-Insensitive Non-Interference)**.** *Let C be a choreography that is well-typed under the judgement $\Delta; \Gamma; \bot \vdash C$. Given two choreographic stores Σ_1, Σ_2, satisfying the low-equivalence relation $\Sigma_1 \equiv^{\Gamma}_{low} \Sigma_2$, suppose there exist terminating executions:*

$$\langle C, \Sigma_1, \mathscr{C} \rangle \Downarrow^{M_1} \Sigma_1' \quad \textit{and} \quad \langle C, \Sigma_2, \mathscr{C} \rangle \Downarrow^{M_2} \Sigma_2'$$

then the resulting stores satisfy the relation $\Sigma_1' \equiv^{\Gamma}_{low} \Sigma_2'$.

Proof outline. Conceptually, the proof proceeds by constructing corresponding executions in the instrumented semantics and then applying a standard progress and preservation style argument [11], where low-equivalence is shown to be invariant along instrumented reductions. We now present the main lemmas necessary to complete the argument:

- *Completeness*: this lemma establishes a correspondence between the reference and the instrumented semantics. In particular, each terminating execution can be *lifted* to an instrumented execution whose resulting stores are *correct* w.r.t. Σ_1', Σ_2', and *well-formed* w.r.t. Γ.
- *Preservation*: the typing judgement is preserved by the semantics, i.e., starting from a well-typed configuration, each reduction step produces a well-typed configuration. Here, a well-typed configuration consists of a *well-typed choreography* and a *well-formed store*. Handling mutually recursive procedures relies crucially on the *well-typed security procedure context* assumption.
- *Unwinding*: under the appropriate typing and well-formedness assumptions, the instrumented semantics preserves low-equivalence of configurations at each reduction step. This lemma motivates the instrumented syntax and semantics so that, while strict syntactic equality may fail to hold stepwise, low-equivalence of instrumented choreographies is preserved.

The proof proceeds by induction on the number of reduction steps in the instrumented executions, which we construct using the *Completeness* lemma. The zero length case follows clearly from the given definition of *Natural Semantics*. The $n + 1$ case follows by combining the *Preservation* lemma (which let us use the induction hypothesis) and the *Unwinding* lemma (which morally proves non-interference step by step).

Note that we only consider pairs of terminating executions; divergence is intentionally ignored, in line with termination-insensitive non-interference.

Full definitions and proofs are available in [6, Chapter 4].

Notably, in concurrent languages, Termination-Insensitive Non-Interference is often insufficient even for always-terminating programs – e.g., the PIN example of Boudol and Castellani [12] is a classic counterexample where the final value of a low-security variable reveals a secret, with no divergence involved. The choreographic language we present structurally prevent such attacks, since it has no shared memory and all synchronisation is through explicit point-to-point communications, making the PIN-style attack pattern inexpressible.

5. Context Reconstruction

We present an algorithm that reconstructs Δ from procedure definitions $\mathscr{C}$ through constraint generation and solving.

Constraint Generation Pass. A *constraint* has the form $\Psi \sqsubseteq p.x$, where Ψ is a symbolic *bound* built from located variables, a distinguished fresh variable η, and lattice operations. We write $[\![\Psi]\!]\,\Gamma\,pc$ for the evaluation of Ψ under an environment Γ and a program-counter label pc. The evaluation of a constraint results in a truth value.

The constraint-generation algorithm $\delta, \eta \vdash C \rhd E$ is defined by recursion on choreographies: it returns a finite set of constraints E, given a constraint context δ (mapping procedure names to constraint sets) and a fresh variable η used to represent bounds on pc. For each procedure $X(\bar{p}) = C$ we traverse C and emit exactly the ordering requirements suggested by the typing rules from Section 3, while keeping expression levels symbolic. For instance, $p.x := e$ generates constraint $\Psi_e \sqcup \eta \sqsubseteq p.x$ where Ψ_e symbolically represents e's security level and η represents the security level of pc. The main difference with the typing rules is the treatment of procedure calls: a call imports the constraint set already associated with the callee in δ, renaming formal parameters to actual arguments. This *constraint forwarding* is also the key intuition behind monotonicity in δ of the algorithm.

Iteration. A single reconstruction pass produces constraints for one procedure body under a given δ. We define an operator $\phi_{\mathscr{C}}$ that maps δ to a new context δ' by running constraint generation once for each procedure body in $\mathscr{C}$ and collecting the resulting constraint sets. Since $\phi_{\mathscr{C}}$ is monotonic, Knaster-Tarski's theorem [13] ensures the existence of a *least* fixed point, denoted $\mu\phi_{\mathscr{C}}$.

Generating Δ. We define $\Delta \triangleq \underline{gen}\ (\mu\phi_{\mathscr{C}})$ where $\underline{gen}\ \delta$ creates a map from procedure and pc to the set of environments that satisfy constraints in δ, formally:

$$\Delta\ (X, pc) \triangleq \{\Gamma \mid \forall(\Psi \sqsubseteq p.x) \in E_X,\ [\![\Psi]\!]\,\Gamma\,pc \sqsubseteq \Gamma\ p.x\}\,,$$

here E_X is the constraint set associated with X in $\mu\phi_{\mathscr{C}}$.

Correctness of Reconstruction

We now show that the generated context is correct with respect to the typing relation. In the following we write C_X to mean $X(\bar{q}) = C_X \in \mathscr{C}$.

Theorem 2 (Well-typedness of the generated procedure context)**.** *Let $\Delta \triangleq \underline{gen}\ (\mu\phi_{\mathscr{C}})$. Then, for every procedure identifier X, program-counter label pc, and environment Γ, if $\Gamma \in \Delta\ X\ pc$, it holds that $\Delta, \Gamma, pc \vdash C_X$.*

Proof outline. The key lemma is a one-step soundness property: if reconstruction produces constraints E for a choreography C under some δ and an environment Γ satisfies E, then C is typable under $\Delta = \underline{gen}\ \delta$ and Γ. Intuitively, constraint reconstruction is just the typing rules *run backwards*: for each construct it emits exactly the flow-inequalities needed as rule premises, so any Γ that satisfies the generated constraints immediately provides the hypotheses to rebuild the typing derivation for C. Finally, let $\delta^\star = \mu\phi_{\mathscr{C}}$; since $\delta^\star$ is a fixed point, for every procedure X the constraint set stored at $\delta^\star\ X$ is exactly the one reconstructed from C_X under $\delta^\star$, so any $\Delta^\star = \underline{gen}\ \delta^\star, \Gamma \in \Delta^\star\ X\ pc$ satisfies the hypotheses of one-step soundness and hence $\Delta^\star, \Gamma, pc \vdash C_X$.

Since all the functions in this section are defined over finite structures, there will always be a terminating algorithm to compute them.

Full definitions and proofs are available in [6, Chapter 5].

6. Related Work

A closely related line of work is due to Lluch-Lafuente et al. [14], who study information-flow control for choreographic programs. Their approach targets *discretionary* information-flow policies, expressed as access-control constraints that regulate which principals may read or update particular data items. They instrument the operational semantics with rich information-flow annotations and develop a sound type system that statically checks compliance of a choreographic program with a given policy. In contrast, we establish the stronger semantic property of termination-insensitive non-interference.

More broadly, information-flow security has a long history, beginning with Denning's lattice model [2] and the seminal formulation of non-interference by Goguen and Meseguer [3]. Volpano et al. [4] introduced one of the first type systems for secure information-flow analysis in an imperative setting, and a substantial body of subsequent work has refined and generalised type-based enforcement mechanisms for sequential languages [15, 8]. These systems provide static, compositional guarantees on both explicit and implicit flows, typically via a program-counter discipline closely related to the one we adopt. Our work builds directly on this line of research. The key difference is that we lift these foundational ideas from sequential, local programs to a distributed setting in which communication and control flow are specified globally as choreographies.

Dynamic and hybrid techniques for information-flow security have also been extensively studied. For instance, language-based runtime enforcement mechanisms [15] can monitor executions and prevent insecure behaviour at run time. More recent work on permissive runtime information-flow control [16] improves precision in the presence of language features such as exceptions. However, purely dynamic mechanisms inherently reason about single concrete executions, whereas non-interference is a relational property over multiple executions. As a result, dynamic enforcement alone cannot generally establish semantic non-interference, and extending such guarantees to globally specified distributed protocols can pose additional challenges, especially for certain classes of implicit flows. In contrast, our approach provides a static, compositional proof of termination-insensitive non-interference for choreographic programs.

Our work also relates to type systems for concurrent calculi and session-based formalisms with security annotations. Bhargavan et al. [17] and Capecchi et al. [18] were among the first to integrate access control and information-flow guarantees into session-typed process calculi, combining protocol structure with security levels. Subsequent work on secure session types and multiparty protocols establishes access-control or information-flow properties at the level of local endpoint processes [19, 20]. The distinction between these approaches and ours mirrors the broader difference between multiparty session types and choreographic programming. Session-based approaches typically verify endpoint implementations against a global type specification, deriving local types used to type-check processes whose internal computations are abstracted by the type. In contrast, we verify a full distributed implementation written as a choreographic program, which specifies not only the communication structure but also the concrete data manipulations and control flow. Endpoint implementations are then obtained by projection. Consequently, non-interference is established once at the global program level and inherited by the projected endpoints by construction.

Security protocol verification tools such as ProVerif [21] and Tamarin [22] provide powerful automated support for checking cryptographic protocols in symbolic models. These approaches excel at modelling cryptographic primitives and attacker capabilities, and can verify sophisticated authentication and secrecy properties. However, they operate over specialised protocol formalisms and establish properties by model checking specific protocol instances against user-specified security goals. They do not aim to provide a type-based, compositional characterisation of non-interference integrated into a programming model for distributed applications. Our approach is complementary. We equip the reference choreographic programming language with a static type system that supports compositional, static type checking and reconstruction, and we prove a general syntactic non-interference theorem. Rather than analysing protocols instance by instance, we establish a language-level guarantee that applies uniformly to all well-typed choreographic programs.

Choreographic programming has been extensively developed as a foundation for correct-by-

construction distributed programming [1]. Its central result is endpoint projection: a choreographic program can be compiled into local endpoint implementations that are correct by construction, guaranteeing properties such as deadlock-freedom and communication safety [23, 24, 25, 26, 27]. Foundational work established the semantic correctness of projection and its connections to multiparty session types and global types [28, 29, 30] and have supported further developments such as model-based synthesis [31], logics for reasoning about choreographic programs [32, 33], and mechanically certified compilation to endpoint code [34, 5, 35, 36]. These theoretical foundations have been realised with the implementation of several choreographic programming languages, including Chor [29], AIOCJ [37] and, more recently, Choral [38], and HasChor [39], which integrate choreographic programming with mainstream object-oriented and functional programming languages and explore advanced features such as higher-order choreographies, location polymorphism, and library-level endpoint projection. Across this body of work, the primary focus has been functional correctness and communication safety whereas our work is the first to address non-interference.

Finally, our proof methodology builds on standard syntactic techniques for establishing type soundness and security properties. We follow the progress and preservation style of Pierce [11] and Wright and Felleisen [9], and we make use of natural semantics in the style of Kahn [10] to formulate termination-insensitive non-interference. The use of a bracketed, instrumented semantics is by now standard in language-based information-flow security [15, 8].

7. Conclusion

We have presented a type-based technique for non-interference in choreographic programs. Our type system combines a policy-parametric security lattice with a program-counter discipline to prevent information leaks through both explicit data flows and implicit control dependencies. Well-typed choreographies satisfy termination-insensitive non-interference, ensuring that high-security data cannot influence low-security observations. Our type system supports procedural choreographies through a procedure context for admissible calling environments. We formalised a constraint-based reconstruction algorithm that computes said context, eliminating the need for manual security annotations.

We see several directions for future work. First, we plan to extend our type system to handle more expressive security policies, including declassification mechanisms that permit controlled information release under specified conditions. Second, we intend to explore other security properties, moving beyond termination-insensitive non-interference. Indeed, *side channels* [40] (e.g., timing, termination, resource usage, cache effects, message sizes, scheduler-dependent behaviour) fall outside this view, unless the semantics and the attacker observation model explicitly make them observable. Our termination-insensitive approach deliberately abstracts from timing and divergence, which suffices for many practical scenarios but does not address all potential covert channels. When side channels matter, one can bring them into scope by enriching the semantics with cost or timing observables and adopting timing-/step-sensitive definitions [41]. Finally, we plan to mechanise the entire development in a proof assistant such as Lean or Rocq. While the pen-and-paper proofs provide confidence in our results, a complete mechanisation would offer machine-checked assurance and eliminate the possibility of subtle errors in lengthy case analyses and inductive arguments. Besides strengthening confidence in our results, successfully completing this mechanisation would provide a foundation for developing certified type checkers and verified compilation tool-chains for secure choreographic programming. We envision these checkers as bridges to bring our theoretical results to practical implementation, e.g., by integrating them within existing choreographic programming frameworks, such as Choral [38].

References

[1] F. Montesi, Introduction to Choreographies, Cambridge University Press, 2023.

[2] D. E. Denning, A lattice model of secure information flow, Communications of the ACM 19 (1976) 236–243.

[3] J. A. Goguen, J. Meseguer, Security policies and security models, in: 1982 IEEE Symposium on Security and Privacy, IEEE, 1982, pp. 11–11.

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

[5] L. Cruz-Filipe, F. Montesi, M. Peressotti, A formal theory of choreographic programming, J. Autom. Reason. 67 (2023) 21. doi:10.1007/S10817-023-09665-3.

[6] M. Bertoni, Mechanized Type-Based Enforcement of Non-Interference in Choreographic Languages, Master's thesis, University of Bologna, 2025. URL: https://amslaurea.unibo.it/id/eprint/36727/.

[7] G. D. Plotkin, The origins of structural operational semantics, The Journal of Logic and Algebraic Programming 60 (2004) 3–15.

[8] A. Myers, Proving noninterference for a while-language using small-step operational semantics (2011).

[9] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and computation 115 (1994) 38–94.

[10] G. Kahn, Natural semantics, in: Annual symposium on theoretical aspects of computer science, Springer, 1987, pp. 22–39.

[11] B. C. Pierce, Types and programming languages, MIT press, 2002.

[12] G. Boudol, I. Castellani, Noninterference for concurrent programs, in: F. Orejas, P. G. Spirakis, J. van Leeuwen (Eds.), Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings, volume 2076 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 382–395. doi:10.1007/3-540-48224-5_32.

[13] A. Tarski, A lattice-theoretical fixpoint theorem and its applications. (1955).

[14] A. Lluch-Lafuente, F. Nielson, H. R. Nielson, Discretionary information flow control for interaction-oriented specifications, in: N. Martí-Oliet, P. C. Ölveczky, C. L. Talcott (Eds.), Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday, volume 9200 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 427–450. doi:10.1007/978-3-319-23165-5_20.

[15] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE Journal on selected areas in communications 21 (2003) 5–19.

[16] A. Bichhawat, V. Rajani, D. Garg, C. Hammer, Permissive runtime information flow control in the presence of exceptions, J. Comput. Secur. 29 (2021) 361–401. doi:10.3233/JCS-211385.

[17] K. Bhargavan, R. Corin, P. Deniélou, C. Fournet, J. J. Leifer, Cryptographic protocol synthesis and verification for multiparty sessions, in: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009, IEEE Computer Society, 2009, pp. 124–140. doi:10.1109/CSF.2009.26.

[18] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, T. Rezk, Session types for access and information flow control, in: P. Gastin, F. Laroussinie (Eds.), CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings, volume 6269 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 237–252. doi:10.1007/978-3-642-15375-4_17.

[19] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, Information flow safety in multiparty sessions, Math. Struct. Comput. Sci. 26 (2016) 1352–1394. doi:10.1017/S0960129514000619.

[20] I. Castellani, M. Dezani-Ciancaglini, J. A. Pérez, Self-adaptation and secure information flow in multiparty communications, Formal Aspects Comput. 28 (2016) 669–696. doi:10.1007/S00165-016-0381-3.

[21] B. Blanchet, Modeling and verifying security protocols with the applied pi calculus and proverif, Found. Trends Priv. Secur. 1 (2016) 1–135. doi:10.1561/3300000004.

[22] S. Meier, B. Schmidt, C. Cremers, D. A. Basin, The TAMARIN prover for the symbolic analysis of security protocols, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 696–701. doi:10.1007/978-3-642-39799-8_48.

[23] F. Montesi, Choreographic Programming, Ph.D. thesis, IT University of Copenhagen, 2013. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

[24] L. Cruz-Filipe, F. Montesi, A core model for choreographic programming, Theor. Comput. Sci. 802 (2020) 38–66. doi:10.1016/J.TCS.2019.07.005.

[25] F. Barbanera, I. Lanese, E. Tuosto, Choreography automata, in: S. Bliudze, L. Bocchi (Eds.), Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings, volume 12134 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 86–106. doi:10.1007/978-3-030-50029-0_6.

[26] S. Jongmans, P. van den Bos, A predicate transformer for choreographies - computing preconditions in choreographic programming, in: I. Sergey (Ed.), Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, volume 13240 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 520–547. doi:10.1007/978-3-030-99336-8_19.

[27] F. Barbanera, I. Lanese, E. Tuosto, A theory of formal choreographic languages, Log. Methods Comput. Sci. 19 (2023). doi:10.46298/LMCS-19(3:9)2023.

[28] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party session, Log. Methods Comput. Sci. 8 (2012). doi:10.2168/LMCS-8(1:24)2012.

[29] M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, in: R. Giacobazzi, R. Cousot (Eds.), The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013, ACM, 2013, pp. 263–274. doi:10.1145/2429069.2429101.

[30] P. Deniélou, N. Yoshida, Multiparty session types meet communicating automata, in: H. Seidl (Ed.), Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings, volume 7211 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 194–213. doi:10.1007/978-3-642-28869-2_10.

[31] M. Autili, D. D. Ruscio, A. D. Salle, P. Inverardi, M. Tivoli, A model-based synthesis process for choreography realizability enforcement, in: V. Cortellessa, D. Varró (Eds.), Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, volume 7793 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 37–52. doi:10.1007/978-3-642-37057-1_4.

[32] M. Carbone, F. Montesi, C. Schürmann, Choreographies, logically, Distributed Comput. 31 (2018) 51–67. URL: https://doi.org/10.1007/s00446-017-0295-1. doi:10.1007/S00446-017-0295-1.

[33] L. Cruz-Filipe, E. Graversen, F. Montesi, M. Peressotti, Reasoning about choreographic programs, in: S. Jongmans, A. Lopes (Eds.), Coordination Models and Languages, volume 13908 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 144–162. doi:10.1007/978-3-031-35361-1_8.

[34] L. Cruz-Filipe, F. Montesi, M. Peressotti, Certifying choreography compilation, in: A. Cerone, P. C. Ölveczky (Eds.), Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings, volume 12819 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 115–133. doi:10.1007/978-3-030-85315-0_8.

[35] J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, M. Norrish, Kalas: A verified, end-to-end compiler for a choreographic language, in: J. Andronick, L. de Moura (Eds.), 13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, August 7-10, 2022, volume 237 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 27:1–27:18. URL: https://doi.org/10.

4230/LIPIcs.ITP.2022.27. doi:10.4230/LIPICS.ITP.2022.27.
[36] A. K. Hirsch, D. Garg, Pirouette: higher-order typed functional choreographies, Proc. ACM Program. Lang. 6 (2022) 1–27. URL: https://doi.org/10.1145/3498684. doi:10.1145/3498684.
[37] M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, J. Mauro, Dynamic choreographies: Theory and implementation, Log. Methods Comput. Sci. 13 (2017). doi:10.23638/LMCS-13(2:1)2017.
[38] S. Giallorenzo, F. Montesi, M. Peressotti, Choral: Object-oriented choreographic programming, ACM Trans. Program. Lang. Syst. 46 (2024) 1:1–1:59. doi:10.1145/3632398.
[39] G. Shen, S. Kashiwa, L. Kuper, Haschor: Functional choreographic programming for all (functional pearl), Proc. ACM Program. Lang. 7 (2023) 541–565. doi:10.1145/3607849.
[40] J. Kelsey, B. Schneier, D. Wagner, C. Hall, Side channel cryptanalysis of product ciphers, in: European Symposium on Research in Computer Security, Springer, 1998, pp. 97–110.
[41] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, M. Emmi, Verifying Constant-Time implementations, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, 2016, pp. 53–70. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.