Enforcing Language Semantics Using Proof-Carrying Data

(extended version)

August 14, 2013

Stephen Chong
Harvard University
chong@seas.harvard.edu

Eran Tromer
Tel Aviv University
tromer@cs.tau.ac.il

Jeffrey A. Vaughan
LogicBlox Inc.
jeff.vaughan@logicblox.com

Abstract

The soundness of language-level reasoning about programs relies on program execution adhering to the language semantics. However, in a distributed computation, when a value is sent from one party to another, the receiver faces the question of whether the value is *well-traced*, i.e., could it have produced by a computation that respects the language semantics? Otherwise, accepting the value may lead to bugs or vulnerabilities.

Proof-Carrying Data (PCD) is a recently-introduced cryptographic mechanism that allows messages in a distributed computation to be accompanied by proof that the message *and the history leading to it* complies with a specified predicate. Using PCD, a verifier can be convinced that the predicate held throughout the distributed computation, even in the presence of malicious parties, and at a verification cost that is independent of the size of the computation producing the value. With a suitable choice of predicate, a program may use PCD to check that values received from the network are well-traced. Unfortunately, previous approaches to using PCD required tailoring a specialized predicate for each application, using an inconvenient formalism and with little methodological support.

This work introduces a novel, PCD-based approach to enforcing language semantics in a distributed computation. We show how to construct a runtime, for an object-oriented language, which ensures that objects received from potentially untrusted parties are *well-traced* with respect to any prescribed class definitions. This means programmers can analyze language-level properties of distributed programs in a trusted setting, and then use the runtime to generically enforce the same properties in the presence of malicious parties, without needing to be aware of the the underlying cryptographic techniques.

Contents

1 Introduction **3**

2 Overview **5**

3 Base Language **8**
- 3.1 Syntax 8
- 3.2 Operational semantics 8
- 3.3 Well-traced values 11

4 Communication Language **12**
- 4.1 Syntax and semantics 13
- 4.2 Well-tracedness 14

5 Modeling Distributed Executions **15**
- 5.1 COCOTRUST 15
- 5.2 Distributed execution graphs 15

6 Proof Carrying Data **21**
- 6.1 Overview of PCD 21
- 6.2 A compliance predicate for well-tracedness 23
- 6.3 Verifying well-tracedness 24
- 6.4 Verifying correctness of COCOCOMM executions 25

7 Extensions **27**

8 Related work **28**

9 Conclusion **30**

Acknowledgments **31**

References **32**

A Auxillary Definitions **34**

B Proof of Theorem 1 **36**

1. Introduction

Language features such as encapsulation and modules enable local reasoning about expressive and useful properties of programs and values. The soundness of such local reasoning relies on execution adhering to the language semantics. However, in a distributed computation, when a value is received from an untrusted party, it is potentially difficult to ascertain that the value was computed according to language semantics. Indeed, the received value may be one that could never result from any valid computation. This could happen due to a malicious attack by the sender, transient errors, trojan horses at the software or hardware levels, or (transitively) corruption in values that the sender itself received from others. Such risks are especially a concern with distributed and outsourced cloud computing.

Consider, for example, object deserialization in type-safe languages. A byte stream received from the network could represent an object value that has the appropriate fields for its type, but with a state that is impossible to reach under the language semantics (i.e., it violates class invariants). Allowing such values into the evaluation is potentially harmful, and renders inapplicable any program analysis that relies on those class invariants. Verifying these invariants is, in general, a hard open problem. Bloch devotes an entire chapter of his book *Effective Java* [9] to the problem of, and manual methods for, safe deserialization of objects. In practice, similar concerns are raised by a secure coding standard [10] (listing this problem as "probable" and "high remediation cost"), and by a study of common weaknesses [31].

We seek to enable sound language-level reasoning about values and computations, for distributed computations with untrusted parties that may violate language semantics. We do so by using a recently proposed cryptographic mechanism, *Proof-Carrying Data* [7, 11, 12], to ensure that object values received from a sender were computed in accordance with language semantics (using any prescribed set of classes), and, transitively, that all values previously received by the sender and used in the computation were similarly computed in accordance with language semantics.

Proof-Carrying Data PCD [7, 11, 12] is a recently-introduced cryptographic mechanism that allows every message in a distributed computation to be accompanied by proof that the message *and the history leading to it* complies with a specified *compliance predicate*. Using PCD, a verifier can be convinced that the compliance predicate held throughout the distributed computation, even in the presence of malicious parties, and at a verification cost that is independent of the size of the computation producing the value.

The PCD compliance predicate can be instantiated with any polynomial-time computable property that is required to hold, locally, at every node of the computation. Once a compliance predicate has been chosen, PCD gives a recipe for generating and verifying proofs of this property during the distributed execution. In this work, we design a PCD compliance predicate that asserts that values were produced in accordance with the semantics of a specific object-oriented programming language, and with respect to any prescribed set of classes in that language.

PCD proofs are *probabilistic* and *computationally sound*: there can exist convincing proofs for false statements, but no efficient algorithm can find such bad proofs with more than negligible probability. In this sense, PCD is similar to other cryptographic mechanisms for integrity, such as digital signatures.

Contribution The key contribution is a novel runtime mechanism for enforcing language semantics in a distributed computation with potentially malicious or faulty participants. This enables sound program analysis: because received values are verified to have been computed in accordance with language semantics, developers can reason about properties and invariants of received values without needing to trust the party that computed the value.

A second contribution of this work is to demonstrate that PCD is an effective means to build trustworthy distributed applications, and allows distributed applications to obtain the benefits of PCD in a generic manner, without application-specific customization.

```
class Puzzle {
  remainingMoves; // Integer, no. moves until White wins.
  board; // Board, State of game

  initialize(b) { // gets a Board b, returns a new Puzzle
    if (! b.whiteWins()) return null;

    return (this with { remainingMoves = 0, board = b });
  }

  rewindMove(m) { // gets a Move m, returns the updated Puzzle
    ... // check m is a legal move on the current board
    return (this with
             { board = this.board.undoMove(m),
               remainingMoves = this.remainingMoves + 1} );
  }

  getBoard { return b; }
}

class Board { ...
  ... // representation of board configuration, and whose turn is next
  undoMove(m) {... } // undo Move m and return the updated Board
  whiteWins() {... } // return 1 iff the board represents a White win
}

class Move { ... } // represents a move by a player
```

Figure 1. Class definitions for chess puzzles, in the COCO language. Comments indicate the intended types of methods and fields. Fields are implicitly private, initialized to **null**, and immutable.

Motivating example Consider a distributed application where a sender creates *helpmate chess puzzles* to be solved by a recipient. A helpmate chess problem is specified as a pair of an integer n and a board position. A solution exists if White and Black can conspire such that White wins in exactly n legal moves. It is easy to construct a solvable puzzle by starting from a White checkmate and "rewinding" valid moves, but it's potentially difficult to decide if a given puzzle has a solution.

Figure 1 sketches class definitions intended to implement this application, in an object-oriented programming language (namely, the COCO language described in Section 3). Class Puzzle encodes helpmate problems following the intuition above: Puzzles are initialized with a Board object where White has won and where remainingMoves = 0, and more interesting puzzles are created via calls to rewindMove.

All fields in these classes are implicitly private and immutable. We can reason at the language level to conclude that an invariant holds: every Puzzle object represents a solvable helpmate problem.

However now consider the distributed setting. How can a recipient know if a received Puzzle is really solvable in the stated number of moves? (Observe that a malicious sender might provide data that deserializes into a well-typed Puzzle object, i.e., one with two fields of the right name etc., that is nevertheless unsolvable.) The receiver might attempt to validate the Puzzle by solving it, but this is, in general, expensive.

In the following we will show how to use PCD to enable our recipient to efficiently verify that the Puzzle was (with overwhelming probability) computed in accordance to with language semantics and the above classes, and, consequently, is solvable.

Outline The rest of this paper is organized as follows. Section 2 gives an informal overview of our construction, and intuition about the underlying cryptographic proof system.

In Section 3 we present COCO, a class-based object-oriented language. We define an object v of class C to be *well-traced*, with respect to a given set of classes, if there is a COCO program using those classes that produces v. This usefully captures the concept that an object was produced by a computation that adhered to language semantics.

In Section 4 we present COCOCOMM, which extends COCO with communication primitives. The semantics of COCOCOMM requires that any value received over a channel must be well-traced. This is sufficient to ensure that any value produced by a COCOCOMM program must itself be well-traced. Thus, a party that uses COCOCOMM to execute its part of a distributed computation is able to use language-level local reasoning about object invariants and other properties of the computation. The challenge, of course, is to enforce COCOCOMM semantics, and specifically: to ensure that all received values are well-traced.

In Sections 5 and 6 we show how to use PCD in the language runtime to efficiently enforce COCO-COMM semantics, i.e., to ensure that all values received could have been computed in accordance with the language semantics, with respect to a given set of classes.

Section 7 discusses extensions to our approach, including how to support more full-fledged languages (with imperative state, concurrency, etc.), and how to achieve stronger proof notions. We discuss related work in Section 8, and conclude in Section 9.

2. Overview

Subsequent sections will formally describe our approach and prove its correctness. This will necessitate a lengthy formal treatment, and hide much of the underlying machinery using the Proof-Carrying Data abstraction. But first, for concreteness, in this section we forgo abstraction and provide a brisk, informal walk-through, focusing on the runtime operation of the approach and intuition for its soundness. This journey will gradually explain our techniques, starting from naive solutions and building towards (an instance of) the general approach, while simplifying or omitting most technical details.

Consider, first, the case of two parties, Alice and Bob. They have agreed upon an object-oriented programming language (e.g., COCO), and upon a class table CT_{cmn} providing a common vocabulary (e.g., our chess puzzle example). Each party also has arbitrary additional local classes, and some local program to evaluate. Bob's program, at some point, receives from Alice an object v of a class C in CT_{cmn}.

All Bob knows about v is its class C. Using modular reasoning, relying on the language semantics and the content of CT_{cmn}, Bob deduces an invariant of C which v must thus fulfill (e.g., the puzzle is solvable). Bob then proves that this invariant implies a desirable property of his program (e.g., it eventually solves the puzzle represented by v). If both programs are loaded into a single **trusted joint runtime** (e.g., a single interpreter running on a trusted platform), which passes v from Alice's program to Bob's, then Bob's property of course holds. It still holds if the programs are separated onto different computers, using **separate but trusted runtime systems** that correctly execute both programs and, whenever Alice sends a value, faithfully serialize it, transmit it to Bob over a network, and deserialize it there.

Suppose, however, that Alice runs an **untrusted sending runtime** that may produce corrupted values due to malice, faults, trojan horses, rogue cloud services, unreliable network, etc. When v is received, the deserialization in Bob's runtime easily ensures that v is *well-formed*, i.e., has the right fields. But for the invariant to hold the value should be moreover *well-traced*, i.e., indeed produced by Alice using some program and classes of the above form. This is not guaranteed, and in general may be infeasible to check

directly (e.g., how would one efficiently check if a chess puzzle is solvable?). And if v is not well-traced, then Bob's desirable property may not hold.

Bob still trusts his own computer (otherwise it is meaningless to discuss the behavior of his program). He replaces his local runtime with a *verifying runtime*, which requires Alice to provide evidence that v is well-traced. Alice (if she is honest) then installs a *proving runtime*. Her runtime first executes her program as usual, until the point where her programs sends v. This may entail nondeterministic choices resolved in arbitrary ways, such as external inputs, human interaction, or randomness; these choices are recorded in a *trace-witness* W. Alice's runtime then packages up everything needed to reproduce v: Alice's local program t, her local classes CT, and W; call this her *local input*, linp. Alice's runtime sends her linp to Bob's runtime, which checks it using a fixed program, called the *compliance predicate* $\mathbb{C}$, which operates in the straightforward manner: given a value v and local input linp structured as above, $\mathbb{C}(v, \mathsf{linp})$ makes sure that CT indeed properly contains CT_{cmn}, and then it replays the whole computation to make sure that v is the correct result of executing program t using classes CT and using trace-witness W to resolve nondeterministic choices. Bob's runtime will agree to deserialize v only if $\mathbb{C}(v, \mathsf{linp})$ accepts. We have thus restored the soundness of Bob's analysis, but using **inefficient verification and long proofs**.

We would like to move the burden of proving back to Alice. To convince Bob that there exists linp that results in v (and thus v is well-traced, and thus v maintains the class invariants), Alice she should runs $\mathbb{C}$ on her own computer, and convince Bob of the following.

Alice's Statement: "There exists some linp *such that* $\mathbb{C}(v, \mathsf{linp})$ *accepts."*

Fortunately, *Probabilistically-Checkable Proofs (PCPs)* [1] allow us to do just that. Alice's runtime first records a step-by-step trace T of $\mathbb{C}$'s execution (note that $\mathbb{C}$ is a fixed, known part of the runtime). It then uses a PCP to encode T into proof π_{PCP}, which is a string of bits with a special error-detection property: there is an efficient algorithm, the *PCP verifier* V_{PCP}, that gets v as input, samples just a handful of locations in the string π_{PCP}, and just by a simple consistency check on these bits (e.g., a few parity checks), decides whether T is a correct trace of an execution of $\mathbb{C}(v, \mathsf{linp})$ for the *given* v and *some* linp. The PCP verifier is *probabilistically sound*, in the sense that if there does not exist linp such that $\mathbb{C}(v, \mathsf{linp})$ accepts, then for any proof string π_{PCP}, the PCP verifier will almost always catch an inconsistency in π_{PCP} and reject. Alice's runtime thus sends v and π_{PCP} to Bob's runtime, and the latter runs V_{PCP}. Alas, the encoded string π_{PCP} is even longer than the trace T it encodes. We have thus attained a **probabilistically-sound efficient verification with long proofs**.

The next improvement is to reduce the proof size, and this can be done using cryptographic machinery. Alice's runtime again produces the PCP proof string π_{PCP} as above. It then summarizes π_{PCP} and commits to it, by computing its cryptographic (Merkle) hash digest r. The hash digest r then determines which locations in π_{PCP} to sample. Alice's runtime creates a new proof string π consisting of the hash digest r, the samples, and cryptographic evidence for consistency between these. It then sends v and π to Bob's runtime. Bob's runtime runs a verifier V_{CSP} to check this. $V_{\mathsf{CSP}}(v, \pi)$ checks the hash consistency and then invokes V_{PCP} to check the samples. Soundness now relies on cryptographic assumptions (e.g., that no efficient algorithm can find colliding inputs to the hash function, or manipulate the hash input to control the choice of samples). Thus, these are *computationally-sound proofs* [30], whose soundness holds only if the adversary is an efficient algorithm. The size of π is merely logarithmic in that of π_{PCP} (and T). We have thus attained **computationally-sound efficient verification with short proofs** for Alice's Statement, and thus for well-tracedness of v. This is the best Bob could hope for.

More generally, similar computationally-sound proofs can **generically compress proofs** for NP statements (i.e., statements of the form "there exists w such that $\Phi(x, w)$ accepts") for a polytime Φ. We will make use of this later.

Let us now extend the scenario to a **chain of untrust**. Consider a third party, Carol, who has her own computer and program. She joins the conversation, using the same CT_{cmn} as a common vocabulary of classes. After receiving v from Alice, Bob's program sends a value v' (derived from v) to Carol. In her turn, Carol runs some COCO program that receives v', and depends on v' being well-traced in order to ensure some property. Carol does not trust Alice or Bob. She thus installs the same verifying runtime described above, using computationally-sound proofs, and expects Bob to produce a suitable proof π' for v'. Bob can compute his local input linp' similarly to Alice, but when he tries to run $\mathbb{C}$, $\mathbb{C}$ notices that at some point the replayed computation receives a value (from Alice)—and this must not be allowed, for fear that the value is not well-traced. Indeed, even if Bob correctly evaluated every step his own program, he may have accidentally (or maliciously) received values that were not well-traced, and in that case, Bob's computation is corrupted and v' may not fulfill its class invariants either (e.g., Bob may have simply forwarded $v' = v$).

We thus require a mechanism for **transitive verification**: Bob's proof π' should convince Carol not merely that Bob executed his own program correctly, but also that he verified that every value he received was well-traced. The combination of these ensures that v' is well-traced.

To realize this we make two extensions. First, the compliance predicate $\mathbb{C}$ is extended with an additional parameter. Now, $\mathbb{C}(v, \mathsf{linp}, \vec{v}_{\mathsf{in}})$, where as before $\mathsf{linp} = (CT, t, W)$, makes sure that CT indeed properly contains CT_{cmn}, and then it replays the computation to make sure that v is the correct result of executing program t using classes CT, using trace-witness W to resolve nondeterministic choices, and—here lies the addition—using $\vec{v}_{\mathsf{in}}$ as the list of received values. This takes care of verifying Bob's own COCO evaluation. In order to also transitively verify that $\vec{v}_{\mathsf{in}}$ are well-traced, we define the following algorithm $\mathbb{M}_{\mathbb{C}}$. On inputs value v, local input linp, received values $\vec{v}_{\mathsf{in}}$ and corresponding proofs $\vec{\pi}_{\mathsf{in}}$, $\mathbb{M}_{\mathbb{C}}(v, \mathsf{linp}, \vec{v}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$ verifies that $\mathbb{C}(v, \mathsf{linp}, \vec{v}_{\mathsf{in}})$ accepts and, moreover, for every input $v_i \in \vec{v}_{\mathsf{in}}$ and corresponding proof $\pi_i \in \vec{\pi}_{\mathsf{in}}$, the aforementioned verifier $V_{\mathsf{CSP}}(v_i, \pi_i)$ accepts. Carol asks Bob to prove the following about v'.

Bob's Statement: "There exists some linp*, received values* $\vec{v}_{\mathsf{in}}$ *and corresponding proofs* $\vec{\pi}_{\mathsf{in}}$ *such that* $\mathbb{M}_{\mathbb{C}}(v', \mathsf{linp}, \vec{v}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}})$ *accepts."*

Bob then convinces Carol, using a computationally-sound proof for this statement. Concretely, Bob's proof π' is a compression of the execution trace of $\mathbb{M}_{\mathbb{C}}(v', \mathsf{linp}', (v), (\pi))$, where v and π are the value and proof he received from Alice. The result is a transitive proof that convinces Carol that v' is well-traced.[1]

Note how, in the above, proofs always "piggybacked" on values: every data value sent was accompanied by a stand-alone proof that attests to its well-tracedness. Carol did not have to go back and talk to Alice; the proof produced by Bob already summarized his history. This idea naturally generalizes to general **distributed computation among untrusted parties**. Consider a distributed system, consisting of any number of parties, communicating in an arbitrary pattern and mutually-distrusting. We can ensure that, jointly, they are following the prescribed language semantics with respect to common classes CT_{cmn}, by having all of them follow a protocol similar to Bob: verify every incoming value, and produce a transitive proof for every sent value, using computationally-sound proofs about (a generalization of) the algorithm $\mathbb{M}_{\mathbb{C}}$. The proofs stay at a constant size, regardless of the length of computation or number of parties, and always "summarize" the relevant history back to the dawn of computation. This is a **Proof-Carrying Data** system for the property of well-tracedness.

In the rest of this paper, we will follow the reverse path. Existing constructions of Proof-Carrying Data (Chiesa and Tromer [11] following Micali [30] and Kilian [26], of which the above is a sketch; and Bitansky et al. [7] which takes an alternative approach) already tackle the cryptographic details and soundness analysis, and encapsulate them as a "cryptographic primitive" which lets any compliance predicate (of

[1] While intuitively appealing, this is far from obvious and requires additional ingredients [43][11].

suitable form) be enforced. We will take these as granted, and *instantiate* them for well-tracedness, for a particular, illustrative object-oriented language, by defining a suitable compliance predicate and proving its requisite properties.

3. Base Language

This section introduces COCO, a **Co**re **C**alculus for **O**bjects. COCO is a class-based object-oriented calculus broadly based on Featherweight Java [24], but modified to model encapsulation and to support a clear, robust definition of well-tracedness. COCO allows only functional update of fields, and all fields are private—i.e., fields of objects of class C may be updated only by code of class C. Private, immutable fields make it sensible to (inutitively) state that an object o of class C is well-traced if o is returned by a method of C or otherwise escapes the scope of such a method.

As described below COCO contains two operators, **escape** and **capture**, which are intended to model the essence of values escaping though a heap or heap-like mechanism. And a non-deterministic choice operator makes it easy to construct and reason about observers of escaped values, even in the presence of non-termination.

3.1 Syntax

The syntax of COCO is intuitively described in the following, and formally specified in Figure 2. The notation $\overline{X}$ indicates a set (without duplicates) of objects in syntactic category X. We use subscripts to refer to individual elements of $\overline{X}$, e.g., X_i.

COCO is a class-based object-oriented language; a class table CT is a partial map from names to class declarations, which define the classes' fields and methods. We write $\mathrm{dom}(CT)$ for the set of class names for which CT provides a definition, and $names(CT)$ for the set of class names used or defined in CT.

Objects have form $\{C \mid \overline{f = v}\}$, where C is the class of the object, and the fields f_i of the object map to values v_i. Operator **new** C creates an object $\{C \mid \overline{f = \textbf{null}}\}$, i.e., all fields initially map to the special value **null**. Notation $s.m(t)$ invokes method m with receiver s and argument t. All methods take one argument and return a value. Fields are immutable and implicitly private: fields may be read only via term **this**.f, and term **this with** $\{\overline{g = s}\}$ creates a new object identical to the object denoted by **this**, except each g_i is mapped to the result of evaluating s_i.

Term $s \mathbin{[]} t$ concurrently evaluates both s and t; if either evaluates to some value v, $s \mathbin{[]} t$ may evaluate to v. We use term $s \mathbin{[]} t$ to provide a limited model of both concurrency and nondeterministic choice.

We provide a simple mechanism for reasoning about observability of values, meant to capture the essential ways by which values can escape the context where they are created (e.g., through imperative state, communication, exceptions or coroutines—none of which exist in COCO). Terms **escape** t and **capture** are used to model values that can escape the dynamic scope of their owner . Term **escape** t evaluates t to a value v, marks v as "escaping" by appending it to a list S of escaped values, and evaluates to v. Term **capture** nondeterministically evaluates to any value in S (i.e., that previously escaped). If a class does not contain any **escape** terms, then the only way a value owned by that class can escape is if the value is passed as an argument to a method (of another class) that may apply **escape** to it, or if the value is returned from a method that may be invoked by arbitrary code. This allows code in a class to temporarily violate object invariants, so long as it can ensure that any and all objects that may escape satisfy object invariants. We use **escape** and **capture** to mark where objects escape and where escaped objects may be used, thus allowing us to avoid directly defining what it means for an object to escape, or defining semantics for object invariants.

Metavariable convention for names

C, D	class names ($\in$ **CName**)	s, t	terms ($\in$ **Term**)
CT	class tables ($\in \mathcal{CT}$)	u, v, o	values ($\in$ **Val**)
f, g	field names	i	integers ($\in \mathbb{Z}$)
m, n	method names	$\oplus$	total integer operations
x, y, z	variable names ($\in$ **Var**)		

Syntax

Class Table
$CT \in \mathcal{CT} = \mathbf{CName} \rightharpoonup \mathbf{CDecl}$

Class Declaration
$\chi ::= \textsf{class}\{\textsf{fields}\,\overline{f};\, \textsf{meths}\,\overline{M}\,\} \in \mathbf{ClassDecl}$

Method Declaration
$M ::= m(x)\,\{\textsf{return}\,t;\}$

Terms

$s, t ::=$	x	Variable	|	$s.m(t)$	Method invocation
|	i	Integer constants	|	$\textsf{this}.f$	Field access
|	$\textsf{this}$	The current object	|	$\textsf{this}\,\textsf{with}\{\,\overline{f = s}\}$	Functional field update
|	$\textsf{null}$	Null value	|	$\textsf{if}\ t\ \textsf{then}\ s_1\ \textsf{else}\ s_2$	If not zero
|	$\textsf{new}\,C$	Object creation	|	$\textsf{let}\,x = s\,\textsf{in}\,t$	Let
|	$t\ \textsf{instanceof}\ C$	Instance-of test	|	$\textsf{escape}\,s$	Escape
|	$\textsf{isnull}\,t$	Null test	|	$\textsf{capture}$	Capture
|	$s \oplus t$	Integer operations	|	$s \,[\,]\, t$	Nondeterministic choice

Values

$o, v ::=$	$\{C \mid \overline{f = v}\,\}$	Object	|	$\textsf{null}$	Null value
|	i	Integer value			

Figure 2. Syntax of COCO

3.2 Operational semantics

Figure 3 defines the small-step operational semantics for COCO. Although mostly straightforward, a few interesting aspects of the semantics are discussed below.

Judgment $CT, \rho \vdash \langle t, S\rangle \rightarrow \langle t', S'\rangle$ indicates that, under class table CT, and environment ρ, term t takes one small step to t', with the list of escaped values growing from S to S'. When a small step evaluation of t evaluates an $\textsf{escape}\,v$ term then $S' = S \bullet [v]$ (where $\bullet$ denotes list concatenation); otherwise $S' = S$. For simplicity, we write $CT \vdash \langle t, S\rangle \rightarrow \langle t', S'\rangle$ when environment ρ is empty. We write $\emptyset$ for the empty list of escaped values. As usual, $CT \vdash \langle s, S\rangle \rightarrow^* \langle t, S'\rangle$ indicates zero or more sequential small steps.

Environments $\rho \in \mathbf{Var} \cup \{\textsf{this}\} \rightharpoonup \mathbf{Val}$ map variables and $\textsf{this}$ to values. We write $\emptyset$ for the empty environment. Environments are modified by $\textsf{let}$ terms and by method invocations. For instance, $\textsf{let}\,x = v\,\textsf{in}\,t$ steps to $\textsf{let}\,x = v\,\textsf{in}\,t'$ by evaluating t in an environment where ρ has been extended to $\rho[x \mapsto v]$.

Evaluation Contexts

$$E ::= [\cdot] \mid \textsf{isnull}\ E \mid E \oplus t \mid v \oplus E \mid \textsf{if}\ E\ \textsf{then}\ s\ \textsf{else}\ t \mid E\ \textsf{instanceof}\ C \mid E.m(t) \mid v.m(E)$$
$$\mid \textsf{this with}\{\,\overline{g = v}, f = E, \overline{g' = t}\} \mid E \,[\!]\, t \mid t \,[\!]\, E \mid \textsf{let}\ x\ =\ E\ \textsf{in}\ t \mid \textsf{escape}\ E$$

$\boxed{CT, \rho \vdash \langle s, S \rangle \to \langle t, S' \rangle}$

$$\frac{CT, \rho \vdash \langle s, S \rangle \to \langle s', S' \rangle}{CT, \rho \vdash \langle E[s], S \rangle \to \langle E[s'], S' \rangle} \qquad \frac{\rho(x) = v}{CT, \rho \vdash \langle x, S \rangle \to \langle v, S \rangle} \qquad \frac{\rho(\textsf{this}) = v}{CT, \rho \vdash \langle \textsf{this}, S \rangle \to \langle v, S \rangle}$$

$$\frac{}{CT, \rho \vdash \langle i_1 \oplus i_2, S \rangle \to \langle i, S \rangle}\ i_1 \oplus i_2 = i \qquad \frac{}{CT, \rho \vdash \langle \textsf{isnull null}, S \rangle \to \langle 1, S \rangle}$$

$$\frac{}{CT, \rho \vdash \langle \textsf{isnull}\ v, S \rangle \to \langle 0, S \rangle}\ v \neq \textsf{null} \qquad \frac{}{CT, \rho \vdash \langle \textsf{if}\ i\ \textsf{then}\ s_1\ \textsf{else}\ s_2, S \rangle \to \langle s_1, S \rangle}\ i \neq 0$$

$$\frac{}{CT, \rho \vdash \langle \textsf{if}\ 0\ \textsf{then}\ s_1\ \textsf{else}\ s_2, S \rangle \to \langle s_2, S \rangle} \qquad \frac{}{CT, \rho \vdash \langle \{C \mid \overline{f = u}\}\ \textsf{instanceof}\ C, S \rangle \to \langle 1, S \rangle}$$

$$\frac{}{CT, \rho \vdash \langle v\ \textsf{instanceof}\ C, S \rangle \to \langle 0, S \rangle}\ v \neq \{C \mid \overline{f = u}\} \qquad \frac{\rho(\textsf{this}) = \{C \mid \overline{f = u}, g = v\}}{CT, \rho \vdash \langle \textsf{this}.g, S \rangle \to \langle v, S \rangle}$$

$$\frac{CT(C) = \textsf{class}\{\textsf{fields}\ \overline{f};\ \textsf{meths}\ \overline{M}\}}{CT, \rho \vdash \langle \textsf{new}\ C, S \rangle \to \langle \{C \mid \overline{f = \textsf{null}}\}, S \rangle} \qquad \frac{\begin{array}{c} v_1 = \{C \mid \overline{f = u}\} \\ CT(C) = \textsf{class}\{\textsf{fields}\ \overline{f};\ \textsf{meths}\ \overline{M}, m(x)\,\{\textsf{return}\ t;\}\} \end{array}}{CT, \rho \vdash \langle v_1.m(v_2), S \rangle \to \langle \textsf{mcall}(t, [\textsf{this} \mapsto v_1, x \mapsto v_2]), S \rangle}$$

$$\frac{CT, \rho' \vdash \langle t, S \rangle \to \langle t', S' \rangle}{CT, \rho \vdash \langle \textsf{mcall}(t, \rho'), S \rangle \to \langle \textsf{mcall}(t', \rho'), S' \rangle} \qquad \frac{}{CT, \rho \vdash \langle \textsf{mcall}(v, \rho'), S \rangle \to \langle v, S \rangle}$$

$$\frac{\rho(\textsf{this}) = \{C \mid \overline{f = u}, \overline{g = v}\ \}}{CT, \rho \vdash \langle \textsf{this with}\{\,\overline{g = v'}\}, S \rangle \to \langle \{C \mid \overline{f = u}, \overline{g = v'}\,\}, S \rangle} \qquad \frac{}{CT, \rho \vdash \langle \textsf{let}\ x\ =\ u\ \textsf{in}\ v, S \rangle \to \langle v, S \rangle}$$

$$\frac{CT, \rho[x \mapsto v] \vdash \langle t, S \rangle \to \langle t', S' \rangle}{CT, \rho \vdash \langle \textsf{let}\ x\ =\ v\ \textsf{in}\ t, S \rangle \to \langle \textsf{let}\ x\ =\ v\ \textsf{in}\ t', S' \rangle} \qquad \frac{}{CT, \rho \vdash \langle v \,[\!]\, t, S \rangle \to \langle v, S \rangle}$$

$$\frac{}{CT, \rho \vdash \langle t \,[\!]\, v, S \rangle \to \langle v, S \rangle} \qquad \frac{}{CT, \rho \vdash \langle \textsf{escape}\ v, S \rangle \to \langle v, S \bullet [v] \rangle} \qquad \frac{v \in S}{CT, \rho \vdash \langle \textsf{capture}, S \rangle \to \langle v, S \rangle}$$

Figure 3. Operational semantics of COCO

To define the operational semantics of method calls, we extend the surface syntax (Figure 2) with a new syntactic form, $\textsf{mcall}(t, \rho)$, which is used to indicate a partially evaluated method call. The extended syntax is shown in Figure 4. In the remainder of the paper, we use "*program*" to mean a COCO term that uses just surface syntax, and does not contain a term of the form $\textsf{mcall}(t, \rho)$.

Terms

$$
\begin{array}{lll}
s, t & ::= & \ldots \\
& | & \textsf{mcall}(t, \rho) \qquad \text{Partially-evaluated method call}
\end{array}
$$

Environments

$$\rho \quad \in \quad \mathbf{Var} \cup \{\textsf{this}\} \rightharpoonup \mathbf{Val}$$

Figure 4. Deep syntax of Coco

Intuitively, term $v.m(u)$ invokes method m on object v, passing value u as an argument. If the class declaration of v's class contains method declaration $m(x)\,\{\textsf{return}\, t;\}$, then $v.m(u)$ steps to $\textsf{mcall}(t, [\textsf{this} \mapsto v, x \mapsto u])$. More generally, term $\textsf{mcall}(t, \rho)$ represents a method invocation that is currently executing, where t is the rest of the method body to execute, and ρ is the environment for the callee. Note that unlike **let** terms, evaluation of **mcall** replaces the current environment instead of extending it; this models the fact that all variables are local to a method.

As mentioned previously, term $\textsf{escape}\, v$ adds v to the list S of escaped values, and **capture** evaluates nondeterministically to a previously escaped value. Note that the list of escaped values only increases during execution.

A small-step evaluation of $s \,[]\, t$ may take a step for either s or t (using appropriate evaluation contexts), or, if either s or t is a value, then it may step to that value.

3.3 Well-traced values

We say that a value v is *well-traced in class table* CT if there is some Coco program that can evaluate to v using a class table that is consistent with CT.

The property of well-tracedness is central to our goal of enforcing language semantics in distributed computation. Intuitively, if value v is well-traced in class table CT, then v could have been produced by a program execution that adhered to the language semantics, using class definitions that are compatible with class table CT, but possibly including additional classes.

To formally define well-tracedness, we first introduce some notation, and define what it means for a value to be well-formed in a class table, and for one class table to extend another.

We must first present some preliminary definitions. Judgment $CT \vdash_{\mathsf{wf}} v$, defined in Figure 5, indicates that v is *well-formed in class table* CT. Intuitively, well-formedness requires that objects are consistent with their class definitions. That is, object $o = \{C \,|\, \overline{f = v}\,\}$ is well-formed if C appears in CT, the fields $\overline{f}$ bound by object o are exactly the fields that the class C has according to CT, and, recursively, the values $\overline{v}$ that o binds to fields are also well formed in CT. Well-formedness (unlike well-tracedness) is directly and efficiently checkable. If an object o is not well formed, then o is inconsistent with class table CT, and use of o may result in stuck computation.

Class table CT extends class table CT' when CT contains all of the class declarations of CT', and possibly some additional class declarations.

Definition 1 (Class table extension)**.** *Class table CT* extends *class table CT' (written $CT \supseteq CT'$) if* $\mathrm{dom}(CT') \subseteq \mathrm{dom}(CT)$ *and for all* $C \in \mathrm{dom}(CT')$, $CT(C) = CT'(C)$.

We can now define formally define well-tracedness:

$\boxed{CT \vdash_{\mathsf{wf}} v}$

$$\frac{\begin{array}{c} CT(C) = \mathbf{class}\{\mathbf{fields}\ f_1, \ldots, f_n;\ \mathbf{meths}\ \overline{M}\ \} \\ CT \vdash_{\mathsf{wf}} v_i \text{ for } i \in 1..n \qquad names(CT) \subseteq \mathrm{dom}(CT) \end{array}}{CT \vdash_{\mathsf{wf}} \{C \mid f_1 = v_1, \ldots, f_n = v_n\ \}}$$

$$\frac{names(CT) \subseteq \mathrm{dom}(CT)}{CT \vdash_{\mathsf{wf}} \mathbf{null}} \qquad \frac{names(CT) \subseteq \mathrm{dom}(CT)}{CT \vdash_{\mathsf{wf}} i}$$

Figure 5. Inference rules for well-formedness

Definition 2 (Well-traced values)**.** *Value v is* well-traced in CT *if* $CT \vdash_{\mathsf{wf}} v$ *and there exist a class table CT' and a program t such that*

$$CT' \supseteq CT \quad \textit{and} \quad CT' \vdash t \rightarrow^* v\ .$$

Here, the class table CT is intended to represent well-known class definitions, such as standard libraries and common application-specific classes (e.g., Puzzle, Board, and Move for the chess puzzle example). The computation that produced the value, however, may use additional classes beyond those defined in CT. For example, the computation may use classes that are defined by another party in the distributed system.

Escaped values The inclusion of **escape** and **capture** terms in COCO allows us to regard as well-traced any object that escapes the dynamic scope of the owning class: if we have a t such that v escapes during evaluation of t, then program $t \,[\!]\,$ **capture** can evaluate to v, and thus v is well-traced. For example, even if the classes in CT do not include any **escape** terms, if their code passes a value v to a method of an arbitrary class, then that could be a class in some CT' (extending CT) which invokes **escape** on v, which means we could write a COCO program that escapes v; hence v is well-traced in CT.

Example Consider again the chess puzzle example from the Introduction. (The pseudo-code used in Figure 1 is close to, but not exactly, COCO syntax.) Suppose that class table CT_{cmn} contains class definitions for common classes, including Puzzle, Board, and Move. Party A creates a chess puzzle by running a program with some class table CT_A, local to her, which extends CT_{cmn}. Class table CT_A might contain proprietary algorithms implemented in classes that are not part of the agreed upon class table between participants. Thus, when party B receives the chess puzzle, the chess puzzle is well-traced in CT_{cmn}, even though CT_{cmn} does not contain all the classes actually used to produce the chess puzzle.

Moreover, party B, upon receiving a well-traced object v of class Puzzle is able to reason locally about the code for classes Puzzle, Board, and Move, and determine that if v.remainingMoves and v.board are non-null, then there really is a solution to the puzzle. So long as v is well-traced, this holds true regardless of the origin of v, even if A manufactured v without executing any COCO program.

4. Communication Language

We are interested in enforcing language semantics in distributed computations. Since the base language COCO models only local computation without interaction, we extend it into the calculus COCOCOMM by adding a communication primitive. A COCOCOMM program represents the local computation of one party of a distributed computation; communication with other parties occurs by sending and receiving values on channels. The semantics of COCOCOMM ensure that values received over a channel must be well-traced

Metavariable convention for names
ch ranges over channel names

Syntax

Terms

$$
\begin{array}{llll}
s, t & ::= & \ldots & \\
 & | & \textsf{\textbf{send}}\ t\ \textsf{\textbf{on}}\ ch & \text{Send} \\
 & | & \textsf{\textbf{recv on}}\ ch & \text{Receive}
\end{array}
$$

Figure 6. Syntax of COCOCOMM (extending COCO)

in a common class table. Thus, language semantics are enforced, and the local reasoning can be used to establish and maintain object invariants.

4.1 Syntax and semantics

COCOCOMM extends the syntax of COCO with terms for sending and receiving values over channels. The syntax extension is given in Figure 6. Term **send** t **on** ch evaluates term t to a value, which is sent over channel ch, and the term evaluates to the value sent. Term **recv on** ch receives a value on channel ch and evaluates to the value received. We require that values sent and received are well-formed in a *common class table*: a set of class names and definitions that communicating parties have agreed upon in advance. The use of a common class table ensures that a value received will be well-formed in the receiver's local class table, even though the local class tables of the sender and receiver may differ. This is analogous to restricting class serialization in Java to only classes from well-known and agreed upon libraries. (We could extend our model to allow mobile code, that is, to allow class definitions to be sent over channels. For simplicity, we refrain from doing so.) We require that the common class table contains only COCO terms. That is, code in the common class table does not send or receive values (although it may invoke code that does so, through the use of call backs).

Judgment $CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{comm}} \langle t, S\rangle \rightarrow \langle t', S'\rangle$ indicates that COCOCOMM term t takes one small step to term t', using class table CT and environment ρ, using common class table CT_{cmn} for communication, with the list of escaped values growing from S to S'. That is, all values sent and received during the computation must be well-formed in CT_{cmn}. We require that class table CT extends the common class table CT_{cmn}, that is, $CT \supseteq CT_{\mathsf{cmn}}$. Note that CT_{cmn} is a COCO class table: the definitions of the classes of CT_{cmn} do not contain any **send** or **recv** terms; CT may, however, contain classes with **send** or **recv** terms. We write $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t, S\rangle \rightarrow^* \langle t', S'\rangle$ to indicate that term t can take zero or more steps to evaluate to term t' (with an empty environment).

Operational semantics for COCOCOMM are similar to that of COCO: for each inference rule in the operational semantics of COCO (Figure 3) we have a corresponding inference rule for COCOCOMM. In addition, we have inference rules for the terms **send** t **on** ch and **recv on** ch, given in Figure 7. The rule for **send** t **on** ch evaluates t to a value, and ensures that the value is well-formed in CT_{cmn}, and adds v to the list of escaped values. The rule for **recv on** ch nondeterministically receives some value v that is well-traced in CT_{cmn}.

Intuitively, if a value received is well-traced in CT_{cmn}, then the recipient can use local reasoning to establish invariants about the received value, that is, can examine the code of the relevant classes in CT_{cmn} in order to reason about the received value. The recipient does *not* need to trust the sender of the value, or be concerned about how the value was produced. The challenge is to enforce the semantics

$$\boxed{CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{comm}} \langle t, S\rangle \rightarrow \langle t', S'\rangle}$$

$$E ::= \ldots \mid \textbf{send } E \textbf{ on } ch$$

$$\frac{CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} v}{CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{comm}} \langle \textbf{send } v \textbf{ on } ch, S\rangle \rightarrow \langle v, S \bullet [v]\rangle}$$

$$\frac{v \text{ is well-traced in } CT_{\mathsf{cmn}}}{CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{comm}} \langle \textbf{recv on } ch, S\rangle \rightarrow \langle v, S\rangle}$$

Figure 7. Semantics of COCOCOMM (extending COCO)

of COCOCOMM, that is, to ensure that all values received are well-traced. We address this in Sections 5 and 6.

Returning to the chess puzzle example from the Introduction, we see that a common class table that contains appropriate versions of the Puzzle, Board, and Move classes would allow a party to know, upon receiving object $\{\mathsf{Puzzle} \mid \mathsf{remainingMoves} = n, \mathsf{board} = v\}$, that the board must have a solution within n moves, even though the party does not necessarily trust the sender of the object, and has no idea how the sender produced the object.

COCOCOMM is a strict extension of COCO: if t is a COCO program that evaluates to value v, then it is also a COCOCOMM program that evaluates to v. The converse does not necessarily hold.

Communication model We do not explicitly model the operation of communication channels. In Section 5.2 we define a model for composing concurrently executing programs so that **send** and **recv** operations are matched up appropriately, but this model is underspecified with respect to whether communication over a channel is point-to-point or broadcast, whether delivery is in-order, whether drops or duplicate deliveries are possible, etc. These details are inconsequential for the present discussion and theorems.

4.2 Well-tracedness

A key observation is that if a COCOCOMM program, executing with local class table CT, allows a value v to esacpe, and v is well-formed in CT, then v is well-traced in CT.

Theorem 1. *Let CT be a* COCOCOMM *class table and let CT_{cmn} be a common class table such that $CT \supseteq CT_{\mathsf{cmn}}$. Let t be a* COCOCOMM *program, t' a term, S a sequence of escaped values, and v a value such that $v \in S$. If*

$$CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t, \emptyset\rangle \rightarrow^* \langle t', S\rangle$$

and $CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} v$ then v is well-traced in CT_{cmn}.

The proof of this theorem is in Appendix B.

Theorem 1 has two key implications. First, **soundness**: a party using COCOCOMM to carry out its part of a distributed computation can be sure that any values produced by that computation (i.e., any values that can escape) could also have been produced by a plain COCO program (with respect to the same CT_{cmn}). That is, the party does not need to be concerned that values received over communication channels can somehow violate properties or invariants that can be established by reasoning about only COCO programs.

The second implication is **completeness**: honest parties can successfully use COCOCOMM to perform a distributed computation, since any value (of a type on CT_{cmn}) that an honest party sends over a channel

$$\overline{CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{trust}} \langle \textbf{recv on } ch, S\rangle \to \langle v, S\rangle}$$

Figure 8. Receive rule for COCOTRUST (modifing COCOCOMM.)

will be well-traced in CT_{cmn}, and can thus be accepted by the receiver of the value. Although we do not directly examine the semantics of multiple COCOCOMM programs communicating with each other, Theorem 1 will be sufficient to show that honest parties can successfully complete a computation without getting stuck at some communication step due to the well-tracedness requirement of **recv**.

Later, we will argue that these soundness and completeness properties are preserved when the assumption of honesty is replaced by cryptographic proofs.

5. Modeling Distributed Executions

The COCOCOMM language enables a party in a distributed computation to reason locally about values received during computation, without trusting the senders of the values. To enforce COCOCOMM semantics, we must ensure that only well-traced values are received. In this and the following section, we show how to enforce COCOCOMM semantics using Proof-Carrying Data (PCD) [7, 11], a cryptographic mechanism for verified computation. The structure of this and the following section is as follows.

We introduce COCOTRUST, which is similar to COCOCOMM except that received values are not required to be well-traced (Section 5.1). We show that if we have a distributed computation where all parties are executing COCOTRUST programs, then received values are in fact well-traced (Section 5.2 and Theorem 2).

We provide an overview of PCD (Section 6.1), and develop a PCD compliance predicate that states that a computation adheres to COCOTRUST semantics (Section 6.2). By instantiating PCD on this compliance predicate, a party can prove that its computation adhered to COCOTRUST semantics, and, transitively, all values it received were also produced by COCOTRUST semantics.

We show that this yields a sound and complete proof system for well-tracedness of values (Section 6.3 and Theorems 3 and 4, using Theorem 2), and, moreover, a sound and complete proof system for COCOCOMM semantics (Section 6.4 and Theorems 5 and 6).

5.1 COCOTRUST

COCOTRUST shares its syntax of terms and evaluation contexts with COCOCOMM, and has a single (but crucial) relaxation in semantics. Judgment $CT, CT_{\mathsf{cmn}}, \rho \vdash_{\mathsf{trust}} \langle t, S\rangle \to \langle t', S'\rangle$ indicates that COCOCOMM term t takes one small step to term t', using class table CT and environment ρ, using common class table CT_{cmn} for communication, with the list of escaped values growing from S to S'. As before, we write $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{trust}} \langle t, S\rangle \to^* \langle t', S'\rangle$ to indicate that term t can take zero or more steps, using COCOTRUST semantics, to evaluate to term t' (with an empty environment).

The inference rules for the operational semantics for COCOTRUST are identical to those for COCOCOMM, with the exception of the rule for receiving values. Whereas COCOCOMM requires received values to be well-traced in the common class table CT_{cmn}, COCOTRUST merely "trusts" that received values will be suitable. The COCOTRUST rule for **recv** is shown in Figure 8; note that there is no premise for the rule, and thus no requirements on the received value v. Intuitively, the simplified rule for **recv** makes it easy to efficiently implement COCOTRUST evaluation.

5.2 Distributed execution graphs

Suppose that we have multiple parties, each of which is executing its own CocoTrust program, with common class table CT_{cmn}, and **recv** operations are matched up with appropriate **send** operations. As long as all parties are honest, i.e., adhere to CocoTrust semantics and CT_{cmn}, all values received by parties will in fact be well-traced and thus CocoComm semantics are achieved.

To model this, we define *distributed execution graphs*, which represent the composition of CocoTrust programs with the sends and receives matched up appropriately. The nodes of a distributed execution graph represent zero or more computational steps taken by one of the parties. Directed edges between nodes indicate either the *sequence* of a single party (which first performs the computation represented by the source node, and then the computation represented by the target node), or *communication* between two parties (i.e., the last computational step of the source node is a **send** v **on** ch term, and v is received by the first step of the target node).

We first informally describe distributed execution graphs, and then present a formal definition, stated in a way that simplifies the subsequent use of PCD. The key observation is that in a distributed execution graph (where all parties are following CocoTrust semantics), all values received are well-traced, and thus CocoComm semantics are achieved.

Nodes of a distributed execution graph Each node in a distributed execution graph has a label of the form $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$ where CT is a class table, t and t' are terms, and $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{trust}} \langle t, S\rangle \rightarrow^* \langle t', S'\rangle$. The trace-witness W allows us to efficiently check that it is indeed the case that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{trust}} \langle t, S\rangle \rightarrow^* \langle t', S'\rangle$. This will be useful when we instantiate PCD to help enforce CocoComm semantics. For our purposes, the trace-witness needs to indicate for any nondeterministic choices in the program how the nondeterminism was resolved. For nondeterministic choice terms $s \,[\!]\, t$, the trace-witness indicates whether the left or right term was chosen (and then contains a trace-witness describing how the subterm was evaluated). For **capture** terms, the trace-witness indicates which previously escaped value is selected.

More formally, a *trace-witness* W is a sequence defined by the following grammar, which indicates, for each step, whether the step was deterministic, or, if it was a nondeterministic choice term, whether the left or right term was chosen (and how the nondeterministic choices of the subterm were evaluated), or, if it was a **capture** term, the index i in the list of escaped values of the value to which the term evaluated.

$$\begin{aligned} U &::= \mathit{Det} \mid \mathit{Left} \cdot U \mid \mathit{Right} \cdot U \mid i \\ W &::= (U)^* \end{aligned}$$

We denote the set of possible trace-witnesses as $\mathcal{W}$.

Edges of a distributed execution graph Edges between nodes of a distributed execution graph indicate either sequence or communication, and are labeled to indicate which. The labels also convey the information required for checking consistency between its source and destination nodes.[2]

A sequence edge between two nodes has label $\mathsf{seq}[CT, t', S']$, indicating that the source node computation used class table CT and the last term in the source node's computation was t' with escaped values S'. The target node represents the continuation of the computation by the same party, so the target node's computation will also use class table CT, and the first term in the target node's computation will be t' with escaped values S'.

A communication edge between two nodes has label $\mathsf{msg}[v, ch]$, indicating that the source node sent value v over channel ch, and value v was received by the target node. Thus, the last term of the source node

[2] Thus, Definition 3 speaks of requirements within a node, and on node vs. its incident edges, but never on node vs. another node. This is essential to the subsequent instantiation of PCD to enforce CocoComm semantics, since no node can observe another except by the messages sent.

$$
\begin{aligned}
F ::= {} & [\cdot] \mid \textsf{isnull}\ F \mid F \oplus t \mid v \oplus F \mid \textsf{if}\ F\ \textsf{then}\ s\ \textsf{else}\ t \\
& \mid F\ \textsf{instanceof}\ C \mid F.m(t) \mid v.m(F) \mid \textsf{send}\ F\ \textsf{on}\ ch \\
& \mid \textsf{this with}\{\, \overline{g = v}, f = F, \overline{g' = t}\,\} \mid F \,[]\, t \mid t \,[]\, F \\
& \mid \textsf{let}\ x\ =\ F\ \textsf{in}\ t \mid \textsf{let}\ x\ =\ v\ \textsf{in}\ F \mid \textsf{mcall}(F, \rho) \mid \textsf{escape}\ F
\end{aligned}
$$

Figure 9. Contexts. Compared to CocoComm evaluation contexts, this adds $\textsf{let}\ x\ =\ v\ \textsf{in}\ F$ and $\textsf{mcall}(F, \rho)$.

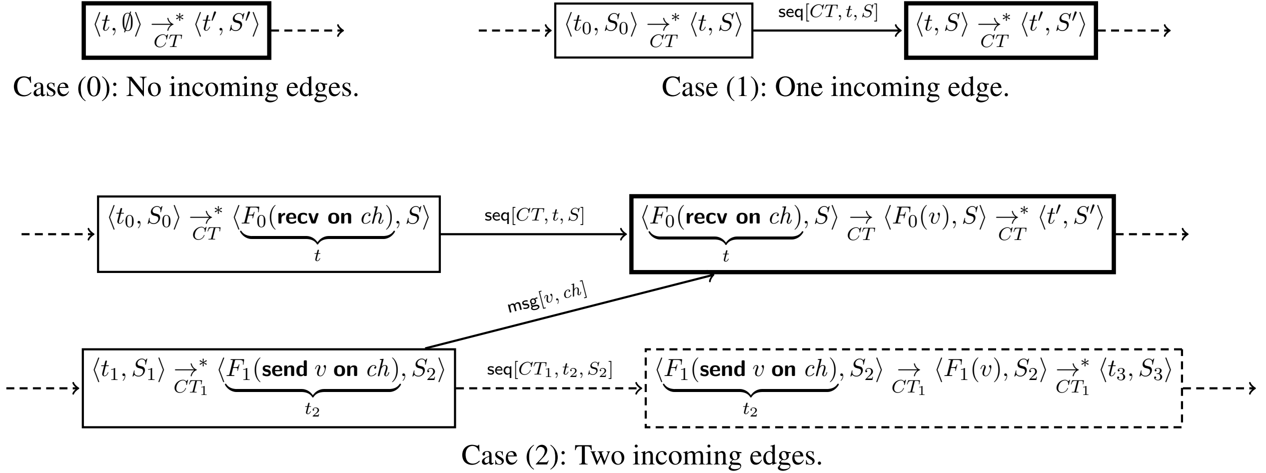

Figure 10. Cases for incoming edges of distributed execution graph nodes. Dashed nodes and edges indicate a typical (but not required) neighborhood. Here, $\langle t, S\rangle \underset{CT}{\rightarrow^*} \langle t', S'\rangle$ is shorthand for $CT, CT_{\textsf{cmn}} \vdash_{\textsf{trust}} \langle t, S\rangle \rightarrow^* \langle t', S'\rangle$.

must have the form $F[\textsf{send}\ v\ \textsf{on}\ ch]$ for some context F; and the first term of the target node must have the form $F'[\textsf{recv on}\ ch]$ for some context F' and then step to $F'[v]$, indicating that the value v was received. Here, contexts F are used to identify **send** and **recv** terms that will be reduced in the next evaluation step. Contexts F include the evaluation contexts of Figures 3 and 7 and also contexts for method calls **mcall** and $\textsf{let}\ x\ =\ v\ \textsf{in}\ t$ expressions. Syntax for these contexts is given in Figure 9.[3]

We restrict the incoming and outgoing edges of nodes to ensure that the graph represents valid computation and communication between parties. For example, we ensure that no node has multiple incoming sequence edges. More precisely, we ensure that for every node $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$, when we consider the incoming edges to the node, there are only three possible cases, enumerated here. (Diagrams illustrating the cases are given in Figure 10.)

(0) No incoming edges. This node represents the start of a computation by a party, where t is the program the party is executing. Escaped values S should be empty.

[3] The CocoComm operational semantics do not include **let** and **mcall** in evaluation contexts E, since these require specialized inference rules and would not be correctly handled by the generic rule for evaluation contexts (first rule in Figure 3). For example, $\textsf{let}\ x\ =\ v\ \textsf{in send}\ x\ \textsf{on}\ ch$ is equivalent to $F[\textsf{send}\ x\ \textsf{on}\ ch]$ where $F = \textsf{let}\ x\ =\ v\ \textsf{in}\ [\cdot]$, but there is no evaluation context E such that it is equivalent to $E[\textsf{send}\ x\ \textsf{on}\ ch]$.

(1) One incoming sequence edge. This node represents the continuation of a local evaluation process: the parent node indicates that the party has evaluated the program up to term t (and escaped values S), and this node continues execution from t (and using the same class table, and escaped values S).

(2) Two incoming edges, consisting of one sequence edge and one communication edge. This node represents the continuation of an evaluation process (via a seq edge) after receiving a value via a msg edge from a sender. Note that the sender will typically continue its execution via another seq edge.

We require that distributed execution graphs are acyclic to ensure that communication is causally ordered: messages aren't received before they are sent. We also require that evaluation of a **recv** term receives exactly one value that was previously sent by a **send** term. We do not otherwise make assumptions about the operations of channels. For example, our model is agnostic as to whether message delivery is in-order or out-of-order, and is agnostic as to whether a sent value should be received at most once (i.e., point-to-point communication), or whether a sent value may be received multiple times along multiple communication edges (e.g., broadcast communication).

Definition of distributed execution graphs Before we formally define distributed execution graphs, we first introduce a helper function, check, that given class table CT, configurations $\langle t, S\rangle$ and $\langle t', S'\rangle$, and trace-witness W, efficiently checks whether $CT \vdash \langle t, S\rangle \to^* \langle t', S'\rangle$. Note that although terms t and t' may contain **send** and **recv** operations, the semantics are COCO, not COCOTRUST. This means that t evaluates to t' without performing any send or receive operations.

Proposition 1 (Existence of check)**.** *There exists a polynomial-time algorithm*

$$\mathsf{check} : \mathcal{CT} \times \big(\mathbf{Term} \times \mathbf{List}(\mathbf{Val})\big) \times \big(\mathbf{Term} \times \mathbf{List}(\mathbf{Val})\big) \times \mathcal{W} \to \{0,1\}$$

such that whenever

$$\mathsf{check}(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$$

it holds that $CT \vdash \langle t, S\rangle \to^* \langle t', S'\rangle$.

Define also a version which skips an inital **send** *if any:*

$$\mathsf{check}'(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = \begin{cases} \mathsf{check}(CT, \langle F[v], S\rangle, \langle t', S'\rangle, W) & \text{if } t = F[\textbf{send } v \textbf{ on } ch] \\ \mathsf{check}(CT, \langle t, S\rangle, \langle t', S'\rangle, W) & \text{otherwise} \end{cases}$$

Trace-witnesses can be produced efficiently.

Proposition 2 (Efficient trace-witness production)**.** *There exists a polytime algorithm that, given a derivation of $CT \vdash \langle t, S\rangle \to^* \langle t', S'\rangle$ yields trace-witness W where* $\mathsf{check}(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$.

Note that the running time of check is polynomial in all of its inputs, including W, which in turn is polynomial in the size of the derivation. However, it independent of the derivation of received messages (indeed, check verifies only plain COCO derivations).

Distributed execution graphs are defined as follows.

Definition 3 (Distributed execution graph)**.** *Given a class table CT_{cmn}, a* distributed execution graph *using CT_{cmn} is a labeled directed acyclic graph fulfilling the following.*

a. Each node's label is of the form $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$ where CT is a class table, t and t' are terms, S and S' are lists of values, and $W \in \mathcal{W}$ is a trace-witness.

b. For each node $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$ and edge with label z_{out} exiting it, exactly one of the following holds:

i. $z_{\mathsf{out}} = \mathsf{seq}[CT, t', S']$

ii. $z_{\mathsf{out}} = \mathsf{msg}[v, ch]$ *for some value* v *and channel* ch*, such that* $t' = F[\textbf{send}\ v\ \textbf{on}\ ch]$ *for some context* F*, and* $CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} v$

c. For each node $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$ *and its list* $\vec{z}_{\mathsf{in}}$ *of incoming edges, exactly one of the following holds:*

(0) $\vec{z}_{\mathsf{in}}$ *is empty and:*

i. CT *extends* CT_{cmn}

ii. $\mathsf{check}'(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$

iii. t *is a program (i.e., it contains only surface syntax terms)*

iv. S *is the empty list* $\emptyset$*.*

(1) $\vec{z}_{\mathsf{in}} = (\,\mathsf{seq}[CT_{\mathsf{E}}, t_{\mathsf{E}}, S_{\mathsf{E}}]\,)$ *such that:*

i. $CT_{\mathsf{E}} = CT$ *and* $t_{\mathsf{E}} = t$ *and* $S_{\mathsf{E}} = S$

ii. $\mathsf{check}'(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$

(2) $\vec{z}_{\mathsf{in}} = (\,\mathsf{seq}[CT_{\mathsf{E}}, t_{\mathsf{E}}, S_{\mathsf{E}}],\ \mathsf{msg}[v, ch]\,)$ *such that:*

i. $CT_{\mathsf{E}} = CT$ *and* $t_{\mathsf{E}} = t$ *and* $S_{\mathsf{E}} = S$

ii. $t = F[\textbf{recv on}\ ch]$ *for some context* F

iii. $\mathsf{check}(CT, \langle F[v], S\rangle, \langle t', S'\rangle, W) = 1$

Distributed execution graphs represent the execution of communicating COCOTRUST (rather than COCOCOMM) programs. Recall, however, that COCOCOMM semantics differ from COCOTRUST semantics only in requiring received values to be well-traced. Since the distributed execution graph explicitly shows the "pedigree" of each received value, it implies that all received values *are* in fact well-traced, and thus, COCOCOMM semantics hold for all nodes in the graph:

Theorem 2. *In a distributed execution graph, for every node labeled* $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$*, it holds that*

$$CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t, S\rangle \to^* \langle t', S'\rangle\ .$$

Proof. By induction on the structure of the graph. Since the graph is acyclic, the induction is well founded. The induction hypothesis is that for any node $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$, there exists a program t_0 such that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \to^* \langle t', S'\rangle$ and moreover, if $t' = F[\textbf{send}\ v\ \textbf{on}\ ch]$ and $CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} v$ then v is well-traced in CT_{cmn}.

We will use the following lemma, which is straightforward from the definitions:

Lemma 1. *For any program* t_0*, terms* t*,*t'*, class tables* CT*,*CT_{cmn}*, lists* S*,*S' *and trace witness* W*: if*

$$CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \to^* \langle t, S'\rangle \quad \textit{and} \quad \mathsf{check}'(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$$

then

$$CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \to^* \langle t', S'\rangle\ .$$

Consider node n, labeled $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$. Assume the inductive hypothesis holds for all nodes that can reach n. To show that there exists a program t_0 such that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \to^* \langle t', S'\rangle$, consider the cases of the incoming edges to n.

(0) The cases of no incoming edges is straightforward, letting $t_0 = t$ and noting condition c(0) in Definition 3 and the definition of check'.

(1) In the case of a single incoming edge ($\mathsf{seq}[CT_{\mathsf{E}}, t_{\mathsf{E}}, S_{\mathsf{E}}]$), we know by condition c(1)i of Definition 3 that $CT_{\mathsf{E}} = CT$ and $t_{\mathsf{E}} = t$ and $S_{\mathsf{E}} = S$, and then by condition b(i) of Definition 3 that this edge exited some parent node of the form $(CT, \langle\cdot,\cdot\rangle, \langle t, S\rangle, \cdot)$. By the induction hypothesis applied to the parent, there exists a program t_0 such that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle t, S\rangle$.
   By condition c(1)ii of Definition 3, $\mathsf{check}'(CT, \langle t, S\rangle, \langle t', S'\rangle, W) = 1$ and thus by Lemma 1, $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle t', S'\rangle$ as required.
(2) In the case of two incoming edges, these are $\mathsf{seq}[CT_{\mathsf{E}}, t_{\mathsf{E}} S_{\mathsf{E}}]$ and $\mathsf{msg}[u, ch]$. Similarly to the above, we deduce from conditions c(2)i and b(i) of Definition 3 that there exists a program t_0 such that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle t, S\rangle$. Moreover, by condition c(2)ii, $t = F[\textbf{recv on } ch]$ for some context F.
   By condition b(ii), the parent of the edge $\mathsf{msg}[u, ch]$ is node of the form $(\cdot, \cdot, \langle F_2[\textbf{send } u \textbf{ on } ch], \cdot\rangle, \cdot)$ for some context F_2, and $CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} u$. Thus, by the induction hypothesis, u is well-traced in CT_{cmn}, and thus by COCOCOMM semantics, $CT, CT_{\mathsf{cmn}} \vdash \langle F[\textbf{recv on } ch], S\rangle \rightarrow \langle F[u], S\rangle$.
   Concatenating these derivations, we get: $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle F[u], S\rangle$. Also, by condition c(2)iii, $\mathsf{check}(CT, \langle F[u], S\rangle, \langle t', S'\rangle, W) = 1$ and thus by Lemma 1, $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle t', S'\rangle$ as required.

There remains to show that if $t' = F[\textbf{send } v \textbf{ on } ch]$ for some v, and $CT_{\mathsf{cmn}} \vdash_{\mathsf{wf}} v$, then v is well-traced. Indeed, since $\textbf{send } v \textbf{ on } ch$ adds v to the list of escaped values, we have $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0 \,[\!]\, \textbf{capture}, \emptyset\rangle \rightarrow^* \langle v, S'\rangle$, and so by Theorem 1, v is well-traced.

□

The key implication of this theorem is that in order to show that a received value is well-traced, it suffices to show that the computation that sent the value (and, transitively, any computation on which the sender depended) adhered to COCOTRUST semantics. Using PCD, we can achieve this, and moreover, achieve it efficiently. In the following subsections, we show how to instantiate PCD to achieve this.

A corollary of this theorem is that in a distributed execution graph, any value sent or received is well-traced.

Corollary 1. *In a distributed execution graph, for every edge labeled* $\mathsf{msg}[v, ch]$*,* v *is well-traced in class table* CT_{cmn}*.*

Proof. Consider the source node of an edge $\mathsf{msg}[v, ch]$, and suppose it is $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$. Term t' must be of the form $F[\textbf{send } v \textbf{ on } ch]$ for some F. By the structure of the graph, and by Theorem 2 there is a program t_0 such that $CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t_0, \emptyset\rangle \rightarrow^* \langle F[v], S' \bullet [v]\rangle$. By Theorem 1 we have that v is well-traced, as required. □

Soundness and completeness of distributed execution graphs We argue informally that distributed execution graphs are both sound and complete. We cannot state the soundness and completeness of distributed execution graphs more formally without first defining the concurrent execution of communicating COCOTRUST programs. It is straightforward to do so (i.e., to choose a suitable semantics for communication channels, and to define semantics for concurrent execution of communicating COCOTRUST programs), but this is inconsequential for our claims thus omitted for brevity.

Distributed execution graphs are sound, in the sense that any distributed execution graph corresponds to some scenario of parties concurrently executing COCOTRUST programs that communicate with each other. They are complete, in the sense that given several concurrently executing COCOTRUST programs that communicate with each other, we can construct a distributed execution graph that represents this concurrent execution. Moreover, a distributed execution graph can be incrementally constructed as a

system of CocoTrust programs executes, using the following process: as the programs execute they may nondeterministically perform the following actions on the distributed execution graph:

1. Add an unlabeled node (representing a prospective derivation)
2. Add an edge labeled $\mathsf{seq}[CT, t, S]$ from a labeled node n to an unlabeled node n' (representing n' continuing evaluation from configuration $\langle t, S \rangle$)
3. Add an edge labeled $\mathsf{msg}[v, ch]$ from a labeled node n to an unlabeled node n' (representing n sending the value v to n' on channel ch)
4. Label an unlabeled node (representing the actual derivation and nondeterministic choices)

This process can be more fully specified to ensure that the resulting distributed execution graph satisfies the appropriate constraints and accurately reflects the execution of the CocoTrust programs.

6. Proof Carrying Data

6.1 Overview of PCD

Proof Carrying Data (PCD), introduced by Chiesa and Tromer [11, 12] and developed by Bitansky et al. [6, 7], is a cryptographic mechanism for ensuring that a given property is maintained at every step of a distributed computation among mutually-untrusting parties. The designated property is specified as a *compliance predicate*, and all messages between parties are accompanied by a proof that the message's data, along with all of the distributed computation leading to that message, satisfies the compliance predicate.

We first recall the key definitions of PCD, which capture the notion of a distributed computation that is compliant with a designated property, and define a PCD system that proves and verifies such compliance. See Bitansky et al. [6] for formal definitions.[4]

Distributed computation transcripts Distributed computations are viewed as directed acyclic multi-graphs with vertex labels and edge labels.[5] Vertices represent the computation of programs, and edges represent messages sent between these programs. Such graphs are called *distributed computation transcripts*.

In our case, an "honest" distributed computation transcript will be a distributed execution graph. Thus, node labels should be of the form $(CT, \langle t, S \rangle, \langle t', S' \rangle, W)$, edge labels should be of the form $\mathsf{seq}[CT, t, S]$ or $\mathsf{msg}[v, ch]$, and the conditions in Definition 3 should hold. But *verifying* these conditions is up to the PCD system.

In a *proof-carrying distributed computation transcript*, every edge label z_i is augmented with another *proof string* label π_i. Figure 11 shows an example.

Compliance A *compliance predicate* $\mathbb{C}$ is a polynomial-time computable predicate for a node in a distributed computation transcript. Intuitively, a compliance predicate is a locally verifiable property: the predicate can be checked using just information that is available at a node. By choosing an appropriate compliance predicate, global properties may hold if all nodes in a distributed transaction graph satisfy the compliance predicate.

More precisely, $\mathbb{C}(z_{\mathsf{out}}; \mathsf{linp}, \vec{z}_{\mathsf{in}})$ observes the local environment of a node in a distributed computation transcript: the list of received inputs $\vec{z}_{\mathsf{in}}$; an (alleged) output z_{out}; and the node's label, or *local input*,

[4] We use the definitions of "publicly-verifiable PCD system" of Bitansky et al. [6, 7] which, compared to Chiesa and Tromer [11], do not assume a secure-token/oracle; directly allow arbitrary node in-degree and out-degrees; and simplify the security quantification. We also make some simplifications, mentioned in subsequent footnotes where pertinent.

[5] The definition of Bitansky et al. [6] requires source and sink nodes to be labeled $\perp$. The restriction is inessential and we remove it here for convenience. Also, we identify $\perp$ with $\emptyset$.

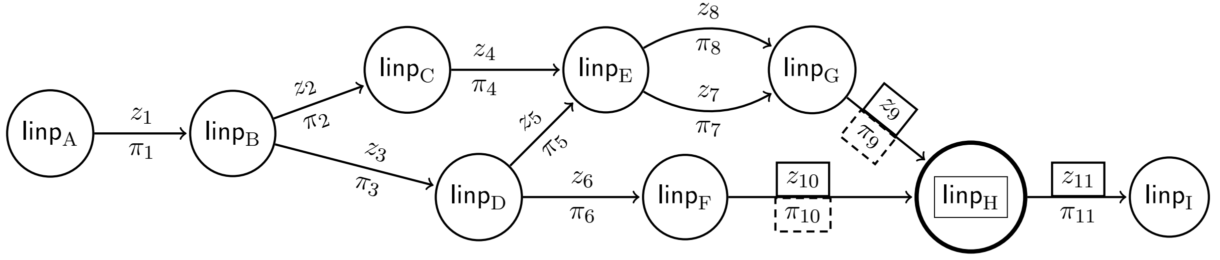

Figure 11. Example of a proof-carrying distributed computation transcript. Node labels ("local inputs") are denoted linp_i, edge labels z_i, and proof strings π_i. The corresponding (non-proof-carrying) distributed computation transcript is given by the same illustration, with proof strings omitted. When checking compliance of the node $\mathsf{linp}_{\mathrm{H}}$, compliance predicate $\mathbb{C}$ checks the values marked by solid boxes. When *proving* compliance of that node, prover $\mathbb{P}_{\mathbb{C}}$ additionally gets the incoming proofs, marked by dashed boxes, and outputs π_{11}.

denoted linp. The label linp contains the executed program and any associated local data (in our case, it will be the local class table CT, configurations $\langle t, S\rangle$ and $\langle t', S'\rangle$, and trace-witness W). Given a distributed computation transcript DCT, we say that node n in DCT, with inputs $\vec{z}_{\mathsf{in}}$ and local input linp, is $\mathbb{C}$-*compliant* if $\mathbb{C}(z_{\mathsf{out}}, \mathsf{linp}, \vec{z}_{\mathsf{in}})$ holds for every output z_{out} of n (see Figure 11). We say that DCT is $\mathbb{C}$-*compliant* if every node in the graph is $\mathbb{C}$-compliant. We say that a string z is $\mathbb{C}$-*compliant* if there exists a $\mathbb{C}$-compliant distributed computation transcript containing an edge labeled z.[6]

PCD syntax and operation A PCD system for a compliance predicate $\mathbb{C}$ is a triple of algorithms $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ as follows:

- *PCD generator* $\mathbb{G}$, given an integer κ as a key size, outputs a key k which will be used by $\mathbb{P}_{\mathbb{C}}$ to generate proofs and by $\mathbb{V}_{\mathbb{C}}$ to verify them.[7]
- *PCD prover* $\mathbb{P}_{\mathbb{C}}$: Let k be a key, let $\vec{z}_{\mathsf{in}}$ be a list of inputs and $\vec{\pi}_{\mathsf{in}}$ be corresponding proof strings, let linp be a local input string, and let z_{out} be an output string. Then $\mathbb{P}_{\mathbb{C}}(k, \vec{z}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}, \mathsf{linp}, z_{\mathsf{out}})$ outputs a proof string π_{out} for the claim that z_{out} is $\mathbb{C}$-compliant.
- *PCD verifier* $\mathbb{V}_{\mathbb{C}}$: Let k be a key, let z_{out} be an output string and π_{out} a corresponding proof string. Then $\mathbb{V}_{\mathbb{C}}(k, z_{\mathsf{out}}, \pi_{\mathsf{out}})$ is meant to accept only if convinced that z_{out} is $\mathbb{C}$-compliant.

Using these algorithms, a distributed computation transcript is dynamically compiled into a *proof-carrying* distributed computation transcript by generating and adding "on the fly" a proof string to each edge (see Figure 11). The process of generating proof strings is defined inductively, starting from the source nodes in the transcript graph. Consider a node n in the transcript, with local input linp, received inputs $\vec{z}_{\mathsf{in}}$ and corresponding proofs $\vec{\pi}_{\mathsf{in}}$, and an output z_{out}. Use prover $\mathbb{P}_{\mathbb{C}}$ to produce a new proof string π_{out} for its output z_{out} (given the inputs of n, their corresponding generated proof strings, the program of n, and its output). Proof strings generated in this way form the additional label on the edges in the resulting proof-carrying transcript.

The triple $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ must satisfy three properties. The first two bound the complexity of proving and verifying, and the third property is computational soundness, discussed next.

[6] In Bitansky et al. [6], the edge labeled z should be the first one entering a sink, but this is inessential since one can consider the subgraph leading to this edge.

[7] For simplicity, we merge the "reference string" σ and "verification state" τ of Bitansky et al. [6] into a single key k.

Completeness and efficient proving The PCD prover can prove true statements, and do so efficiently. Whenever it is indeed the case that a given distributed computation transcript is $\mathbb{C}$-compliant, one can use the PCD prover $\mathbb{P}_{\mathbb{C}}$ to generate proof strings for each message, and all of these proof strings will be accepted by the PCD verifier. The generation of these proofs runs in time that is polynomial in the key size κ and the time it took to run $\mathbb{C}$ at every node in the distributed computation transcript.[8]

Efficient verification Proof strings generated by the PCD prover have length polynomial in the key size κ, and are *efficiently verifiable* by the PCD verifier: $\mathbb{V}_{\mathbb{C}}(k, z_{\mathsf{out}}, \pi_{\mathsf{out}})$ runs in time polynomial in κ and the length of the string z_{out}. In particular, the proof length and verification time are independent of how long it took to evaluate $\mathbb{C}$ and run $\mathbb{P}_{\mathbb{C}}$.[9]

Soundness [10] It is computationally infeasible to prove false statements. This is expressed as follows. Let $\tilde{\mathbb{P}}$ be any efficient cheating prover, i.e., a deterministic algorithm running in time polynomial in the key size κ.[11] After a key $k \leftarrow \mathbb{G}(1^\kappa)$ is randomly generated, $\tilde{\mathbb{P}}(k)$ outputs some non-compliant string z along with a proof string π, and tries to use π to falsely convince $\mathbb{V}_{\mathbb{C}}$ that z is compliant. Then, we are guaranteed that $\mathbb{V}_{\mathbb{C}}(k, z, \pi)$ will accept this false claim with negligible probability $\kappa^{-\omega(1)}$ (i.e., smaller than $1/p(\kappa)$ for any polynomial p). Formally:

Definition 4 (PCD soundness)**.** $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ *is* sound *if for every efficient cheating-prover algorithm* $\tilde{\mathbb{P}}$ *and key size* $\kappa \in \mathbb{N}$*:*

$$\Pr_{k \leftarrow \mathbb{G}(1^\kappa)}\left[\begin{array}{cc} & (z, \pi) \leftarrow \tilde{\mathbb{P}}(k) \\ \wedge & \mathbb{V}_{\mathbb{C}}(k, z, \pi) = 1 \\ \wedge & z \textit{ is not } \mathbb{C}\textit{-compliant} \end{array}\right] \leq \kappa^{-\omega(1)}$$

Construction of PCD PCD systems can be constructed based on standard cryptographic assumptions (collision-resistant hashing schemes and signature schemes), when the parties in the distributed computation have access to a trusted *oracle* (e.g., a secure hardware token or a trusted network service) that signs its inputs [11]. See Section 2 for an overview of this PCP-based construction.

The special case of PCD for a single transmitted message, called *Computationally-Sound Proofs* [30] or *Succinct Non-interactive ARgument of Knowledge (SNARK)*, can constructed without oracles under a large class of assumptions [5]. SNARKs have been recently implemented for C programs [3] and a restricted subset thereof [36], using approaches based on knowledge-of-exponent assumptions [23][28][20][8], in the relaxed "preprocessing" sense, where $\mathbb{G}$ is is allowed to run for time polynomial in that of $\mathbb{C}$. These are practical for very small programs, and research efforts are underway to improve efficiency and implement the PCP-based approach [2]. Using bootstrapping [7], such SNARKs (in principle) suffice to construct PCD systems without oracles, for a restricted but large set of distributed execution graph structures.

[8] The formal definition of completeness requires a precise description of the inductive process of proof generation, and is thus straightforward but long; see Bitansky et al. [6, §5.2].

[9] For simplicity of presentation, we assume here that the potential adversary's power is greater than all "honest" computations that will be proven, and the key size κ is chosen large enough to be secure against such adversaries. Since the adversary's power is at most $\mathrm{poly}(\kappa)$, the length of honest computation can be upper-bounded by $B = \kappa^{\log \kappa}$ (or any other superpolynomial subexponential function), and thus the $\log B$ terms in the proof length and verifier complexity expressions of Bitansky et al. [6] are dominated by κ and can be ommitted.

[10] Soundness is implied by the stronger "proof of knowledge" proved in Bitansky et al. [6, 7], Chiesa and Tromer [11]. For simplicity, in the following we discuss mere soundness, but our construction also offers proof of knowledge (see Section 7).

[11] Or more, generally, a family of circuits, one for each κ, whose size is polynomial in κ.

6.2 A compliance predicate for well-tracedness

We shall use PCD to prove well-tracedness of values sent by a system of communicating COCOCOMM programs. A common class table CT_{cmn} will be fixed in advance, and every value v sent during an (honest) execution will accompanied by a proof that v is well-traced in CT_{cmn}.

Consider a system of communicating COCOCOMM programs using common class table CT_{cmn}, that sends value v from one party to another on channel ch. If this computation is honest, then by the completeness of distributed execution graphs, there is a distributed execution graph DEG with an edge labeled $\mathsf{msg}[v, ch]$. We shall define a PCD compliance predicate $\mathbf{C}_{\mathsf{cmn}}$ that enables efficiently proving the existence of such a DEG.

The compliance predicate $\mathbf{C}_{\mathsf{cmn}}$ will be applied to the nodes of a distributed computation transcript that is *allegedly* a distributed execution graph. The role of $\mathbf{C}_{\mathsf{cmn}}$ is to check that the nodes in the transcript indeed fulfill the constraints defined in Definition 3, i.e., that each node's label describes a correct COCO-TRUST derivation that is consistent with the incoming and outgoing edges' labels.

Definition 5 (Compliance predicate enforcing CT_{cmn})**.** *For a* COCO *class table* CT_{cmn}*, the* compliance predicate enforcing CT_{cmn} *is denoted* $\mathbf{C}_{\mathsf{cmn}}$. *Given local input* linp*, inputs* $\vec{z}_{\mathsf{in}}$ *and (alleged) output* z_{out}*,* $\mathbf{C}_{\mathsf{cmn}}(z_{\mathsf{out}}, \mathsf{linp}, \vec{z}_{\mathsf{in}})$ *accepts if:*

1. linp *is the form* $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$ *where* CT *is a class table,* t *and* t' *are terms,* S *and* S' *are lists of values, and* $W \in \mathcal{W}$ *is a trace-witness, and for these:*
2. *Condition b(i) or b(ii) from Definition 3 holds, and*
3. *Condition c(0), c(1), or c(2) from Definition 3 holds.*

Lemma 2. *Let* CT_{cmn} *be a* COCO *class table. Then every* $\mathbf{C}_{\mathsf{cmn}}$*-compliant distributed computation transcript is also a distributed execution graph using* CT_{cmn}*, and vice versa.*

The above is obvious from the definitions. Thus:

Lemma 3. *Let* CT_{cmn} *be a* COCO *class table, let* v *be a value, and* ch *a channel. If* $z = \mathsf{msg}[v, ch]$ *is* $\mathbf{C}_{\mathsf{cmn}}$*-compliant then* v *is well-traced in* CT_{cmn}*.*

Proof. As z is $\mathbf{C}_{\mathsf{cmn}}$-compliant, there exists a $\mathbf{C}_{\mathsf{cmn}}$-compliant distributed computation transcript DCT containing an edge labeled z. By Lemma 2, DCT is a distributed execution graph using CT_{cmn}, and thus by Corollary 1, v is well-traced in CT_{cmn}. □

6.3 Verifying well-tracedness

By plugging compliance predicate $\mathbf{C}_{\mathsf{cmn}}$ into a PCD construction, we obtain a PCD system $(\mathbb{G}, \mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}, \mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}})$. This PCD system is both sound and complete for efficiently verifying well-tracedness of values. This means that given a well-traced value v, it is possible to construct a proof that v is well-traced, and if we accept a proof that v is well-traced, then it is indeed well-traced (up to a negligible probability of error). This is key to enforcing COCOCOMM semantics, which we will see in Section 6.4.

Soundness To verify well-tracedness of a value v with a proof π, we run $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(k, \mathsf{msg}[v, ch], \pi)$, where k is the key generated by $\mathbb{G}$ (the channel ch does not matter, but is syntactically necessary since $\mathbb{V}_{\mathbb{C}}$ expects to verify edge labels of distributed execution graph). Soundness of the PCD system means that if an efficient cheating-prover algorithm generates a string $z = \mathsf{msg}[v, ch]$ claiming that v is well-traced, along with an alleged proof π for this (as before ch does not matter), but v is actually not well-traced, then $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(z, \pi)$ will accept only with negligible probability (smaller than any $1/\mathrm{poly}(\kappa)$). Formally:

Theorem 3 (Soundness of PCD for well-tracedness)**.** *For any* COCO *class table* CT_{cmn}*, any efficient cheating-prover algorithm* $\tilde{\mathbb{P}}$*, and every key size* κ*:*

$$\Pr_{k \leftarrow \mathbb{G}(1^\kappa)} \left[\begin{array}{l} \quad (z, \pi) \leftarrow \tilde{\mathbb{P}}(k) \\ \wedge \;\; \mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(k, z, \pi) = 1 \\ \wedge \;\; \exists v, ch : z = (\mathsf{msg}[v, ch]) \\ \qquad\qquad \wedge \;\; v \textit{ is not well-traced in } CT_{\mathsf{cmn}} \end{array} \right] \leq \kappa^{-\omega(1)}$$

Proof. The soundness of the PCD system $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$, according to Definition 4, means that whenever the verifier accepts (z, π), it's true that z is $\mathbf{C}_{\mathsf{cmn}}$-compliant (up to negligible probability). By Lemma 3, if $z = \mathsf{msg}[v, ch]$ then v is well-traced in CT_{cmn} (up to negligible probability). □

Completeness The PCD system is complete in that if we have a well-traced value with a known derivation, then we can efficiently produce a proof for its well-tracedness:

Theorem 4 (Completeness of PCD for well-tracedness)**.** *Let* CT_{cmn} *be a* COCO *class table, and let* v *be a well-traced value in* CT_{cmn}*, witnessed by* $CT' \vdash \langle t, \emptyset \rangle \to^* \langle v, S \rangle$ *where* CT' *is a* COCO *class table such that* $CT' \supseteq CT_{\mathsf{cmn}}$*. Consider the* COCOTRUST *execution*

$$CT', CT_{\mathsf{cmn}} \vdash_{\mathsf{trust}} \langle \textbf{send } t \textbf{ on } ch, \emptyset \rangle \to^* \langle v, S \bullet [v] \rangle$$

Let W *be the trace-witness corresponding to the above derivation (by Proposition 2) and let* $\mathsf{linp} = (CT, \langle \textbf{send } t \textbf{ on } ch, \emptyset \rangle, \langle v, S \bullet [v] \rangle, W)$*.*

Then for any key size κ *and key* $k \leftarrow \mathbb{G}(1^\kappa)$*, the proof* $\pi = \mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}(k, \emptyset, \emptyset, \mathsf{linp}, \mathsf{msg}[v, ch])$ *is always accepted:*

$$\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(k, \mathsf{msg}[v, ch], \pi) = 1.$$

Moreover, π *is computed in time polynomial in the size of the derivation* $CT \vdash \langle t, \emptyset \rangle \to^* \langle v, S \rangle$*.*

Proof. Consider the graph DEG consisting of a node labeled linp, with no input edges, and single outgoing edge labled z going to another node labeled $(CT_{\mathsf{cmn}}, \langle \textbf{recv on } ch, \emptyset \rangle, \langle v, \emptyset \rangle, \emptyset)$. It is readily verified to be a distributed execution graph using CT_{cmn}, and thus by Lemma 2, DEG is also a $\mathbf{C}_{\mathsf{cmn}}$-compliant distributed computation transcript. The claim then follows from the completeness of the PCD system. Efficiency follows from $\mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}$ being polynomial-time in its inputs, and the size of W being polynomial in the size of the derivation. □

6.4 Verifying correctness of COCOCOMM executions

The PCD system $(\mathbb{G}, \mathbb{P}_{\mathbb{C}}, \mathbb{V}_{\mathbb{C}})$ is able to verify the correct execution of a number of parties each of which is executing its own COCOCOMM program. That is, we can use the PCD system to build a runtime system for COCOCOMM that correctly enforces semantics even when non-well-traced values are received from rogue parties.

Soundness The soundness of the verification procedure does not depend on the runtime. A party who performs a COCOTRUST evaluation, and uses $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}$ to verify every received value, can rest assured that (except with negligible probability) the evaluation follows the stricter COCOCOMM semantics: the probability that $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}$ accepts all received values, and yet the evaluation violates COCOCOMM semantics, is negligible (smaller than any $1/\mathrm{poly}(\kappa)$). Formally:

Theorem 5 (Soundness of verifying COCOCOMM)**.** *Consider a* COCOTRUST *derivation:*

$$CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{trust}} \langle t, S \rangle \to^* \langle t', S' \rangle$$

Let $(v_i, ch_i)_{i=1}^{\ell}$ be the list of values, and corresponding channels, received in this derivation. Then for any efficient cheating-prover algorithm $\tilde{\mathbb{P}}'$, and every key size κ:

$$\Pr_{k \leftarrow \mathbb{G}(1^\kappa)} \left[\begin{array}{l} \quad \pi_i \leftarrow \tilde{\mathbb{P}}'(k, i) \text{ for } 1..\ell \\ \wedge\ \mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}\big(k,\, \mathsf{msg}[v_i, ch_i]\, \pi_i\big) = 1 \text{ for } 1..\ell \\ \wedge\ \neg\ CT, CT_{\mathsf{cmn}} \vdash_{\mathsf{comm}} \langle t, S \rangle \to^* \langle t', S' \rangle \end{array} \right] \leq \kappa^{-\omega(1)}$$

Proof. The only difference between the semantics of CocoComm and CocoTrust is the premise of the **recv** rule, so the above probability is bounded by:

$$\leq \Pr_{k \leftarrow \mathbb{G}(1^\kappa)} \left[\begin{array}{l} \quad \pi_i \leftarrow \tilde{\mathbb{P}}'(k, i) \text{ for } 1..\ell \\ \wedge\ \exists i \in 1..\ell : \mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}\big(k,\, \mathsf{msg}[v_i, ch_i]\, \pi_i\big) = 1 \\ \qquad\qquad \wedge\ v_i \text{ is not well-traced in } CT_{\mathsf{cmn}} \end{array} \right]$$

Invoking the union bound on the events for separate i:

$$\leq \sum_{i=1}^{\ell} \Pr_{k \leftarrow \mathbb{G}(1^\kappa)} \left[\begin{array}{l} \quad \pi_i \leftarrow \tilde{\mathbb{P}}'(k, i) \\ \wedge\ \mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}\big(k,\, \mathsf{msg}[v_i, ch_i]\, \pi_i\big) = 1 \\ \wedge\ v_i \text{ is not well-traced in } CT_{\mathsf{cmn}} \end{array} \right]$$

and invoking Theorem 3 (with $\tilde{\mathbb{P}}$ derived from $\tilde{\mathbb{P}}'$ by hardcoding i and $z = \mathsf{msg}[v_i, ch_i,]$):

$$\leq \sum_{i=1}^{\ell} \kappa^{-\omega(1)} \leq \kappa^{-\omega(1)}$$

□

Proof generation Generating the above proof requires an inductive process accompanying the whole computation. Every time a party in the computation sends a message (msg edge) or pauses its derivation to receive a message (seq edge), it will run the PCD prover $\mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}$ to produce a proof that the computation was correct so far. Thus, we modify the process for incremental generation of a distributed execution graph (Section 5.2) to ensure that each edge also has a proof associated with it.[12] Intially, we run the PCD generator $\mathbb{G}$ with a sufficiently large key size κ to obtain a key $k \leftarrow \mathbb{G}(1^\kappa)$. Then we run the following *proof generation* process: every time a new edge is added (Operations 2 and 3), with edge label z_{out}, exiting a node labeled $\mathsf{linp} = (CT, \langle t, S \rangle, \langle t', S' \rangle, W)$ whose incoming edges are labeled $\vec{z}_{\mathsf{in}}$ with associated proofs $\vec{\pi}_{\mathsf{in}}$, we generate the proof $\pi = \mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}(k, \vec{z}_{\mathsf{in}}, \vec{\pi}_{\mathsf{in}}, \mathsf{linp}, z_{\mathsf{out}})$ and associate π with the edge. (See Section 2 for a more concrete but less precise account.)

Completeness The proofs constructed as above are indeed accepted by the PCD verifier:

Theorem 6 (Completeness of verifying CocoComm)**.** *Let DEG be a distributed execution graph using* Coco *class table CT_{cmn}. Consider an edge e in DEG, labeled z, from node n to node n'. Consider any key size κ and key $k \leftarrow \mathbb{G}(1^\kappa)$. Let π be a proof associated with e by the above proof generation process. Then π is accepted: $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(k, z, \pi) = 1$.*

Moreover, the running time of the proof generation process for edge e is polynomial in the size of the derivation in n.

Proof. Consider DEG', the subgraph of DEG consisting of n', and all nodes that can reach n' (which of course includes n). DEG' is also a distributed execution graph using CT_{cmn}, and is thus a $\mathbf{C}_{\mathsf{cmn}}$-compliant

[12] For a general formal treatment, see the ProofGen process in Bitansky et al. [6].

distributed computation transcript. Acceptance of π thus follows from the completeness property of PCD (observing that the aforementioned proof generation process is essentially identical to the ProofGen process of Bitansky et al. [6]).

Let the label of n be $(CT, \langle t, S\rangle, \langle t', S'\rangle, W)$; then efficiency follows from $\mathbb{P}_{\mathbf{C}_{\mathsf{cmn}}}$ being polynomial-time in its inputs, and the size of W being polynomial in the size of the derivationto which it attests. □

Remarks In the above soundness definitions, the received values are independent of the PCD key k. The stronger property of **adaptive soundness**, where v_i and ch_i (and thus the resulting derivation) are allowed to depend on k, holds as well (due to the adaptive soundness of the underlying PCD system).

A natural optimization is the case where, in a distributed computation, there exists a **clique of trusted parties** Q that trust each other, but not others. The parties in Q can then verify messages received from outside Q, but omit verification of messages within Q, and still deduce that their (various) derivations follow COCOCOMM semantics.

7. Extensions

Implementation of our approach for nontrivial programs is not yet practical, since sufficiently-efficient implementations of the underlying PCD machinery is not yet available (see Section 6.1). Still, the fundamental possibility of PCD systems motivates the theoretical question of what program properties and language semantics can be thus enforced; and the rapid recent progress in PCD implementations raises the prospect of practical usefulness.

COCO is an intentionally minimal calculus, intended to capture essential language properties and demonstrate matching them to the PCD formalism, while maintaining simplicity. However, our techniques can be extended to handle more expressive language features, and to provides stronger guarantees for consumers of verified values.

Imperative state, including a heap, can be added in a straightforward manner. State is threaded through the operational semantics of the language (as with the list of escaped values), and sent values are accompanied by a heap, which includes all locations that are transitively reachable from the value. When a value and heap are received, locations in the received heap are renamed to ensure disjointness from the current heap, and then the received heap is unioned with the current heap. The definition of distributed execution graphs is extended to enforce state consistency.

With the addition of state, we would no longer require the use of **escape** and **capture** terms to help define well-tracedness, since the language would now contain another, more standard, mechanism by which values may escape the dynamic scope of an owning class.

Concurrency is already modeled by the nondeterministic choice operator $s \,[\!]\, t$, although threads are not guaranteed to run to completion. This can be remedied by adding a concurrent composition operator. Additional mechanisms for synchronization and communication between threads could also be added.

Evolution of values from known past ones (e.g., showing that the computation to produce value v' used value v) is a stronger property than well-tracedness (which allows an arbitrary semantically-allowed history). For example, consider a protocol for a *chess game* (as opposed to the aforementioned *chess puzzle*), where White sends value v describing a board, subsequently receives value v' supposedly representing the board after Black's move, and wishes to ensure that v' is actually the result of a computation that starts with v and legally moves a single Black piece to produce v'.

Such properties can be enforced using mechanisms within the programming language, and do not require modifications to the language, or to the runtime mechanism for enforcing well-tracedness. One approach is to ensure that classes in CT_{cmn} contain a field that is an append-only log of past states (or a cryptographic compression thereof, using hash functions), and the field is updated appropriately by methods of the class. For example, an object representing a chess board may contain a log of all previous

chess boards that led up to it. This ensures that the object itself is recording its evolution, and well-tracedness suffices to allow reasoning about the evolution of the value.

Another approach is to add a field id to a class, and the value of this field is only modified by method setId(r) { **return** (**this with** {id = owf(r)}); } where owf is a cryptographic one-way hash function. Given an object v with a particular value n for the field id, v was produced either by evolution from another object v' with id equal to n, or by a computation that knows a value i such that owf(i)= n. For example, White sets the id field of a board v by calling setId(r) with a large random integer r, and upon receiving v', verifies that v'.getId()=v.getId(). Either v' evolved from v, or Black found a preimage of owf, which is by definition infeasible.

Enforcing encapsulation by the language semantics, along with enforcing of those semantics, enable local language-level reasoning about received values. However, in Java, it is difficult to enforce encapsulation, due to reflection. In order for our technique to be useful, we anticipate that it will be necessary to restrict the use of reflection and any other mechanisms that may violate encapsulation, or otherwise prevent local reasoning. Indeed, many sophisticated Java language analyses are sound only in the absence of reflection.

Note that in our setting we use encapsulation to enforce integrity: ensuring that values of fields of objects are consistent with using the public interface of the object. We are not able to use encapsulation to enforce confidentiality, since a malicious party receiving an object could always violate language semantics to *examine* the values of fields of that object (e.g., by peeking into the interpreter's in-memory representation of values). PCD will only detect violation of language semantics if the malicious party attempts to send a non-well-traced value.

Interaction with the external environment can be modeled in COCO by treating inputs as nondeterministic choices. However, this does not *enforce* that choice resolution is "truly external." To do so we could add input operators to COCO, and make the PCD compliance predicate verify inputs are cryptographically signed by suitable input devices, parties in the environment, etc.

Proof of knowledge is a security property of computational proof systems that is stronger than mere soundness. At times, there is a difference between proving that value v is well-traced and proving that one *knows* a program that (efficiently) produces v. This is especially true in programs that use cryptography. For example, consider the class BobPayment which sets its field amount to an integer i only when a method BobPayment.check is called with arguments i and u, where u is a digital signature on i that verifies under Bob's (hard-coded) signature verification key. Suppose Alice sends Bob the value $v = \{\mathsf{BobPayment} \mid \mathsf{amount}{=}1000000\}$ and a proof that v is well-traced. Should Bob be convinced that he owes her $1M? Clearly a valid signature on "1000000" *exists*, so v is trivially well-traced regardless of whether Alice actually received the signature. However Alice cannot *efficiently compute* the signature by herself, so if she *efficiently* produced v following COCO semantics, then Bob should indeed be convinced. Our proof system indeed (nontrivially) supports reasoning about the computational complexity of producing values. PCD systems offer, beyond mere soundness, the guarantee of *proof of knowledge* [6], which implies essentially the following: whenever the verifier accepts, $\mathbb{V}_{\mathbf{C}_{\mathsf{cmn}}}(k, \mathsf{msg}[v, ch], \pi) = 1$, the (possibly cheating) prover that produced π actually "knows" the full derivation of the value v and "could have" written it down with similar complexity.

Zero-knowledge is another useful property of proof systems, meaning essentially the following: when the verifier accepts, all it learns is that the value is well-traced, and nothing else about its derivation. In particular, if the sender had secrets, they are not revealed by the proof he produces — as desired, e.g., in the chess puzzle example of Section 1. Our approach attains this when invoked using suitable PCD systems [6] or SNARKs [3][36].

8. Related work

Automatic program partitioning Jif/Split [47, 48] automatically partitions single-threaded programs annotated with security types [38] into a program distributed across potentially mutually distrusting host machines. The partitioning ensures that if computation of a value is declared to be independent of a given host, then that host is unable to corrupt or influence the computation of the value. This enables sound reasoning at the language-level of abstraction about the distributed computation: the value computed by the distributed system will be the same as the value computed by the single-threaded program unless one or more of the hosts upon which the computation depends (as indicated by the security-type annotations) is malicious. In Jif/Split, if a host is corrupted, then no computation they perform can be trusted. By contrast, our system allows trust to be gained in computation performed by a corrupted or malicious host: values received from hosts are checked to ensure they are well-traced, which enables sound reasoning about properties that the value must satisfy. However, our system is not suitable for restricting the flow of information in a distributed system, and is thus, is not suitable for enforcing confidentiality requirements.

Enforcing languge semantics Our work ensures that values received over a network were produced according to COCOCOMM semantics, thus enabling sound reasoning about such values at the level of language abstractions, even in the presense of malicious parties. We do not know of existing work exploring this problem.

However, much previous work seeks to enforce language-level semantics in different settings, typically on a single execution platform, or against different classes of faults.

Safe **deserialization** of objects, as discussed in Section 1, is extensively discussed in Bloch's book on practical Java programming [9, Chapter 11]. Pointing out that "serialization is an extralinguistic mechanism for creating objects", Bloch presents several attacks that cause objects to be deserialized into a state that cannot be reached through their public interfaces (i.e., attacks that create non-well-traced objects). Although Bloch describes methods for defensive programming to avoid these attacks, the solutions are manual and error-prone. Similar concerns are raised in CERT's Secure Coding Standard for Java entries SER06-J and SER07-J [10], in Common Weakness Enumeration entry CWE-502 [31], and the references therein.

Numerous works seek to enforce **memory safety** in C and C-like programs, including CCured [34], Cyclone [22, 25], and SAFECode [17, 18]. SAFECode in particular states that their mechanism enables certain analyses to be sound (points-to graph, call graph, and a subset of type information).

Perry et al. [37] present a type system that can detect **transient hardware** faults (a single local corruption) in assembly programs, by maintaining redundant copies of the computations and enforcing their consistency.

Work on **certified compilation** (e.g., [4, 13, 14, 27]) seeks to ensure that program semantics is preserved as the program is is compiled.

Ensuring integrity Various language-based techniques have attempted to ensure the *integrity* of parts of computational processes or the values they create. For instance, automated program partitioning [48] allows an entire distributed computation to written as a single program with security types that is compiled such that only sufficiently trusted hosts may handle integrity-sensitive data. Likewise, evidence based audit [44] allows programs to automatically verify the that incoming values are endorsed by trusted principals. Techniques including proof-carrying code (PCC) [33] and dependently typed programming [16] combine values (which may represent data or code) with machine-checkable proof objects that demonstrate the validity of propositions about the values, independent of their provenance.

These techniques require a programmer to reason about a level of security types, propositions, or proofs, as well as the main computation language. By contrast, PCD allows a programmer to reason entirely at the level of COCOCOMM, and without making implicit or explicit trust assumptions about remote parties.

Such systems are complementary to PCD. PCD can be composed with PCC [11], and dependent types, by using PCD to enable compression, constant-time verification and zero-knowledge privacy for the PCC or dependent-type proofs.

Foreign function interfaces Foreign function interfaces (FFI) allow programs written in a high-level language to call code written in another language (typically C or native machine code). It is available in many popular languages, for performance and interoperability reasons. Typically, the callee can return arbitrary values to the caller, potentially violating the semantics and invariants of the high-level language (i.e., producing values that are not well-traced). Furr and Hoster [19] describe a static analysis approach to this problem, based on cross-language type-inference. Google Native Client [32, 46] restricts the native-code callee, using software fault isolation, to a certain memory region and set of instructions; however, it does not reason about the values output by the callee.

Our approach to proving well-tracedness offers an alternative, in principle: the high-level language can require values produced by FFI calls to be accompanied by proofs of well-tracedness. However, this incurs runtime overhead, which seems unattractive for performance-motivated uses of FFI.

Remote attestation Remote attestation [15, 35] uses a hardware root of trust, typically based on a Trusted Platform Module (TPM), along with cryptography, to let a target machine convince an appraiser/verifier machine that the target machine's state fulfills some properties (e.g., that it is presently running a specific, unmodified and suitably-configured software stack).

Remote attestation can be used to claim well-tracedness of values by attesting that they were produced by executing a given program on top of a given operating system. This attestation is convincing if the appraiser trusts the TPM hardware root of trust,[13] the correct functioning of the hardware platform (e.g., CPU and memory), and the correctness of (requisite properties of) the operating system. Our PCD-based approach removes all of these assumptions.

Similarly to our approach, the BIND [41] and Flicker [29] systems aim to provide *fine-grained attestation*, attesting only to code directly relevant to producing an output. They remove the trust in the operating system, using CPU support for secure execution and late launch. However, trust in the hardware remains inherent. Notably, BIND supports transitive proofs, similarly to PCD.

Unlike most prior works, which focus on low-level properties such as enforcing execution of specific or signed software, we focus on enforcing language semantics, requiring that values produced by a target machine are well-traced.

PCD can be an alternative to hardware-based remote attestation, in scenarios where the latter is used to enforce *integrity* or *correctness* of a value computed by an untrusted target machine. However, PCD cannot express or enforce properties of the target machine as a whole, and thus cannot enforce *secrecy* properties, such as preventing data copying in Digital Rights Management applications.

Verifiable computing The Proof-Carrying Data mechanism falls into the broad area of verifiable computing, which has been extensively studied from a cryptographic perspective. Supporting general programs requires the full power of PCD systems whose construction was discussed in Section 6.1. The single-message case requires SNARKs, likewise discussed. However, if we relax the requirements to allow the verifier to send queries back to the prover (instead of proofs merely piggy-backing on existing messages), then *interactive proof systems* become pertinent, including recent highly-optimized implementations (see Vu et al. [45] and the references therein). If one further relaxes the efficiency requirement on the verifier, then many *secure multiparty computation* and *secure function evaluation* techniques become relevant. In the setting of simultaenously verifying many claims, one can use effiicent *batching arguments* (see Setty

[13] The Pioneer system [39] aims to remove the explicit hardware root of trust, but depends on extensive knowledge of the precise hardware, can be verified only locally, and is subject to some attacks [15].

et al. [40] and the references therein). Some restricted classes of computation, such as those expressible as low-depth circuits, have especially efficient proof systems [21] which were recently implemented (see Thaler [42] and the references therein).

9. Conclusion

This paper presents a novel mechanism to enforce language semantics in a distributed computation with potentially malicious participants. Using the cryptographic mechanism of Proof-Carrying Data (PCD) [7, 11], we ensure that values received must be *well-traced*: they could have been produced by a program execution consistent with the language semantics. Thus, developers can reason about properties and invariants of received values, without trusting the parties that provide the values and without being aware of the underlying cryptographic mechanism.

Acknowledgments

We thank are indebted to Alessandro Chiesa, Daniel Genkin and Noam Rinetzky for their insightful comments.

This work was supported by the Check Point Institute for Information Security; by the Israeli Ministry of Science and Technology, by the Israeli Centers of Research Excellence I-CORE program (center 4/11); by the National Science Foundation under Grant No. 1054172, and by the Air Force Office of Scientific Research under Award No. FA95501210262.

References

[1] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, 1991.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete efficiency of probabilistically-checkable proofs. In *Proceedings of ACM symposium on Symposium on theory of computing*, pages 585–594. ACM, 2013.

[3] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of CRYPTO 2013*, LNCS. Springer, to appear.

[4] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, pages 97–108, 2009.

[5] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of Innovations in Theoretical Computer Science 2012*, pages 326–349. ACM, 2012.

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, Report 2012/095, 2012. `http://eprint.iacr.org/2012/095`.

[7] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of ACM symposium on Symposium on theory of computing*, STOC '13, pages 111–120. ACM, 2013. ISBN 978-1-4503-2029-0.

[8] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[9] J. Bloch. *Effective Java, Second Edition*. Addison-Wesley, Boston, 2nd edition, 2008.

[10] CERT. The CERT oracle secure coding standard for Java, entries SER06-J and SER07-J, 2013. URL `https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java`.

[11] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In A. C.-C. Yao, editor, *ICS*, pages 310–331. Tsinghua University Press, 2010. ISBN 978-7-302-21752-7.

[12] A. Chiesa and E. Tromer. Proof-carrying data: Secure computation on untrusted platforms. *The Next Wave: the NSA's review of emerging technologies*, 19(2):40–46, 2012. URL `http://www.nsa.gov/research/_files/publications/next_wave/TNW_19_2_ProofCarryingData_Chiesa_Tromer.pdf`.

[13] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM.

[14] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 93–106, New York, NY, USA, 2010. ACM.

[15] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[16] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[17] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the Twenty First ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[18] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[19] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 30 (4):18:1–18:63, Aug. 2008. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/1377492.1377493`.

[20] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

[21] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for Muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, 2008.

[22] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[23] J. Groth. Short non-interactive zero-knowledge proofs. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 341–358, 2010.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[26] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, 1992.

[27] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[28] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[29] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[30] S. Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[31] MITRE Corporation. Common weakness enumeration, CWE-502, 2013. URL `http://cwe.mitre.org/data/definitions/502.html`.

[32] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.

[33] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[34] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[35] B. Parno, J. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. SpringerBriefs in computer science. Springer, 2011. ISBN 9781461414605.

[36] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[37] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 42–53, New York, NY, USA, 2007. ACM Press.

[38] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[39] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, Oct. 2005.

[40] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th EuoroSys Conference*, EuroSys '13, pages 71–84, 2013.

[41] E. Shi, A. Perrig, and L. V. Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, Washington, DC, USA, 2005. IEEE Computer Society.

[42] J. Thaler. Time-optimal interactive proofs for circuit evaluation. *CoRR*, abs/1304.3812, 2013.

[43] P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 1–18, 2008.

[44] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*, pages 177–191, 2008. Extended version available as U. Pennsylvania Technical Report MS-CIS-08-09.

[45] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[46] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93, 2009. .

[47] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, New York, NY, USA, Oct. 2001. ACM Press.

[48] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003. IEEE Computer Society.

A. Auxillary Definitions

We write $names(CT)$ for the set of class names used or defined in CT. Collecting class names mentioned in syntax (via $names$) is straightforward because there are no binding occurences of class names and no need to avoid capture. We overload $names$ to return the class names used in terms, environments, class declarations, and sets of values.

Definition 6 ($names$)**.**

$$
\begin{aligned}
names(\textbf{new}\, C) &= \{C\} \\
names(t\ \textbf{instanceof}\ C) &= names(t) \cup \{C\} \\
&\ \vdots \\
names(i) &= \emptyset \\
names(\textbf{null}) &= \emptyset \\
names(\{C \mid \overline{f = v}\,\}) &= \{C\} \cup \Big(\bigcup\nolimits_{(f=v)\in\overline{f=v}} names(v)\Big) \\
names(\rho) &= \bigcup_{x\in\mathrm{dom}(\rho)} names(\rho(x)) \\
names(m(x)\,\{\textbf{return}\, t;\}) &= names(t) \\
names\left(\begin{array}{r}\textbf{class}\{\ \mathsf{fields}\ \overline{f};\\ \mathsf{meths}\,\overline{M}\ \}\end{array}\right) &= \bigcup_{M\in\overline{M}} names(M) \\
names(CT) &= \bigcup_{C\in\mathrm{dom}(CT)} \{C\} \cup names(CT(C)) \\
names(\{v_1,\ldots,v_n\}) &= names(v_1) \cup \cdots \cup names(v_n)
\end{aligned}
$$

We write $e[D/C]$ to denote the replacement of class name C with class name D in expression e. As before, we overload this notation to also define substitution for environments, class declarations, class tables, and sets of values.

Definition 7 ($\cdot[\cdot/\cdot]$)**.**

$$\begin{aligned} C[D/C] &= D \\ E[D/C] &= E \ \textit{where } C \neq E \end{aligned}$$

$$\begin{aligned} (\textsf{\textbf{new}}\ E)[D/C] &= \textsf{\textbf{new}}\ (E[D/C]) \\ (t\ \textsf{\textbf{instanceof}}\ E)[D/C] &= (t[D/C])\ \textsf{\textbf{instanceof}}\ (E[D/C]) \\ &\ \vdots \end{aligned}$$

$$\begin{aligned} i[D/C] &= i \\ \textsf{\textbf{null}}[D/C] &= \textsf{\textbf{null}} \\ \{E \mid \overline{f = v}\,\}[D/C] &= \{E[D/C] \mid \overline{f = (v[D/C])}\,\} \end{aligned}$$

$$\rho[D/C] = \lambda x.(\rho(x)[D/C])$$

$$CT[D/C] = \lambda E. \begin{cases} (CT(C))[D/C] & E = D \\ (CT(E))[D/C] & E \in \mathrm{dom}(CT) \setminus \{C, D\} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

$$\{v_1, \ldots, v_n\}[D/C] = \{v_1[D/C], \ldots, v_n[D/C]\}$$

We define what it means for one environment to extend another.

Definition 8 ($\supseteq$)**.** *We write $\rho_2 \supseteq \rho_1$ when for any $x \in \mathrm{dom}(\rho_1)$ it is the case that $\rho_2(x) = \rho_1(x)$.*

B. Proof of Theorem 1

Given an evaluation $CT \vdash \langle t, S\rangle \to^* \langle t', S'\rangle$, it is possible to rename any class C to a new class name D without otherwise modifying the evaluation.

Lemma 4 (Renaming)**.** *Suppose*

$$CT, \rho \vdash \langle t, S\rangle \to^* \langle t', S'\rangle$$

and

$$D \notin names(CT) \cup names(t) \cup names(\rho) \cup names(t') \cup names(S) \cup names(S') \ .$$

Then, for any class name C, it is the case that

$$CT[D/C], \rho[D/C] \vdash \langle t[D/C], S[D/C]\rangle \to^* \langle t'[D/C], S'[D/C]\rangle.$$

Proof. by induction on the reduction relation. □

Given an evaluation of a program, it is possible to add additional classes to the class table without changing the evaluation.

Lemma 5. *Suppose $CT_2 \supseteq CT_1$ and $CT_1 \vdash \langle t, \emptyset\rangle \to^* \langle t', S\rangle$. Then $CT_2 \vdash \langle t, \emptyset\rangle \to^* \langle t', S\rangle$*

Proof. By a straightforward induction on $CT_1 \vdash \langle t, \emptyset\rangle \to^* \langle t', S\rangle$. □

We are now ready to prove Theorem 1, which states that if value v esacpes during evaluation of a CocoComm program and v is well-formed with respect to common class table CT_{cmn}, then v is well-traced in CT_{cmn}.

Proof of Theorem 1. Suppose we have a CocoComm program t using classtable CT and during an evaluation of t, the value v escapes and values $u_1, \ldots, u_n$ are received. Intuitively, since each value received is well-traced in CT_{cmn}, there are Coco terms $s_1, \ldots, s_n$ and class tables $CT_1, \ldots, CT_n$ such that $CT_i \supseteq CT_{\mathsf{cmn}}$ and $CT_i \vdash \langle s_i, \emptyset \rangle \to^* \langle s'_i, S_i \rangle$ such that $u_i \in S_i$.

We will first construct a class table CT^* such that for all i we have a program $\overline{s}_i$ such that $CT^* \vdash \langle \overline{s}_i, \emptyset \rangle \to^* \langle \overline{s'}_i, \overline{S}_i \rangle$ such that $u_i \in \overline{S}_i$.

For each i, we can rename the classes in s_i and CT_i using Lemma 4 to acquire terms $\overline{s}_i$ and class tables $CT'_i \supseteq CT_{\mathsf{cmn}}$ such that $\mathrm{dom}(CT'_i) \cap (\mathrm{dom}(CT'_j) \cup CT) = \mathrm{dom}(CT_{\mathsf{cmn}})$ for all $i \neq j$, and $CT'_i \vdash \langle \overline{s}_i, \emptyset \rangle \to^* \langle \overline{s'}_i, \overline{S}_i \rangle$ such that $u_i \in \overline{S}_i$.

Let $CT^* = CT \cup \bigcup CT'_i$. Then by Lemma 5 for any i it holds that $CT^* \vdash \langle \overline{s}_i, \emptyset \rangle \to^* \langle \overline{s'}_i, \overline{S}_i \rangle$ such that $u_i \in \overline{S}_i$.

Recall that t is a CocoComm program such that value v escapes during evaluation, and v is well-formed in CT_{cmn}. We must show that v is well traced. We now translate CocoComm program t to a Coco program that replaces all **recv on** ch terms with term **capture** and replaces any term **send** s **on** ch with **escape** s. Moreover, we compose the translation with terms that will mark values $u_1, \ldots, u_n$ as escaping. More precisely we translate t to the following program, which we denote t^*.

$$t^* \equiv \overline{s}_1 \,]\!]\ldots]\!]\, \overline{s}_n \,]\!]\, [\![t]\!]$$

where $[\![\cdot]\!]$ is a function such that

$$[\![\textbf{send } s \textbf{ on } ch]\!] = \textbf{escape } [\![s]\!]$$
$$[\![\textbf{recv on } ch]\!] = \textbf{capture}$$

and is otherwise homomorphic in the struture of terms. Note that t^* is a program that uses only Coco terms (i.e., it does not contain any **send** or **recv** operations).

Moreover, we translate the class table CT^* in a similar way, denoted $[\![CT^*]\!]$. More precisely $[\![CT^*]\!]$ is a class table with the same domain as CT^* and if for class C,

$$CT^*(C) = \textbf{class}\{\textbf{fields } \overline{f};\ \textbf{meths } \overline{m(x)\,\{\textbf{return } t;\}}\,\}$$

then

$$[\![CT^*]\!](C) = \textbf{class}\{\textbf{fields } \overline{f};\ \textbf{meths } \overline{m(x)\,\{\textbf{return } [\![t]\!];\}}\,\}.$$

First note that $[\![CT^*]\!] \vdash \langle t^*, \emptyset \rangle \to^* \langle [\![t]\!], S^* \rangle$ such that $\{u_1, \ldots, u_n\} \subseteq S^*$.

We now show that evaluation of $\langle t, \emptyset \rangle$ with class table CT under CocoComm semantics is in lockstep with evaluation of $\langle [\![t]\!], S^* \rangle$ with class table $[\![CT^*]\!]$ under Coco semantics. We proceeds by induction on the length of the evaluation, with the inductive hypothesis being that if the current CocoComm term is s then the current Coco term is $[\![s]\!]$. For each small step, we proceed by induction on the derivation

$$CT, CT_{\mathsf{cmn}}, \emptyset \vdash_{\mathsf{comm}} \langle s, S \rangle \to \langle s', S' \rangle,$$

to show that if $S' \subseteq S'''$ then

$$[\![CT^*]\!], \emptyset \vdash \langle [\![s]\!], S'' \rangle \to \langle [\![s']\!], S''' \rangle$$

and $S \subseteq S''$. The only interesting case is when the redex is **recv on** ch, and the received value is u_i for some u_i that is well-traced in CT_{cmn}. In the translated version, the redex is **capture**, and the current set of escaped values contains at least $u_1, \ldots, u_n$. Thus, we can step to u_i as required.

Finally, since v is a value, we have $[\![v]\!] = v$, and so we have a program t^* such that

$$[\![CT^*]\!] \vdash \langle t^*, S \rangle \rightarrow^* \langle t', S \rangle$$

for some S such that $v \in S$, and thus v is well-traced, as required. □