Troupe programming language user guide*

April 6, 2022

*Please send your comments and requests for clarifications to aslan@cs.au.dk

CONTENTS

1 INTRODUCTION TO TROUPE

Troupe is a programming language for concurrent and distributed programming with dynamic information flow control. Troupe is a research language, and as such is intended as a playground for research in information flow control.

1.1 BACKGROUND AND INFLUENCE

The design of Troupe is influenced by a number of programming languages and systems. With respect to security, our design draws heavily on the systems such as Fabric/Jif, LIO, and FLAM. With respect to concurrency, the design draws on the systems such as Erlang, Cloud Haskell, and Concurrent ML. Finally, our syntax is SML-like.

1.2 INTENDED AUDIENCE

This guide is intended for researchers and graduate-level students[1] interested in Troupe. We assume that the reader is familiar with basic functional programming and the core concepts of language-based information flow control such as noninterference.

2 SYSTEM ARCHITECTURE

This section describes the basic architecture of the Troupe system and an overview of the programming model.

2.1 TROUPE ARCHITECTURE

Two key concepts in Troupe's architecture are *processes* and *nodes*. A process is the primary unit of computation. Processes are lightweight and communicate with each other using message passing. Troupe processes run on Troupe nodes. A node is the primary unit of trust and corresponds to an instance of the Troupe runtime. Each node has a unique network identifier, and all communication between nodes is point-to-point encrypted using standard techniques.

To enforce information flow control in a decentralized fashion, Troupe combines the notions of standard security levels and *trust* between nodes. There are no special requirements on the underlying label model other than the standard requirements of the distinguished bottom and top elements, denoted $\bot$ and $\top$, operators for the least upper bound and the greatest lower bound, denoted $\sqcup$ and $\sqcap$ respectively, and the security ordering $\sqsubseteq$.

Troupe nodes decide for themselves how much they trust other nodes. Trust is specified via security levels. Every node fully trusts itself, corresponding to trust level $\top$. Trust levels of selected few nodes are specified through runtime configuration. All other nodes have trust level $\bot$. Such nodes are assumed to perform no security monitoring on their end. In particular, nodes that do not run Troupe runtime also have trust level $\bot$ (communication with such nodes is possible for as long as they adhere to the serialization protocol). When communicating with $\bot$-trusted nodes, all data

[1] You have the authors' sympathies if your instructor makes you read this.

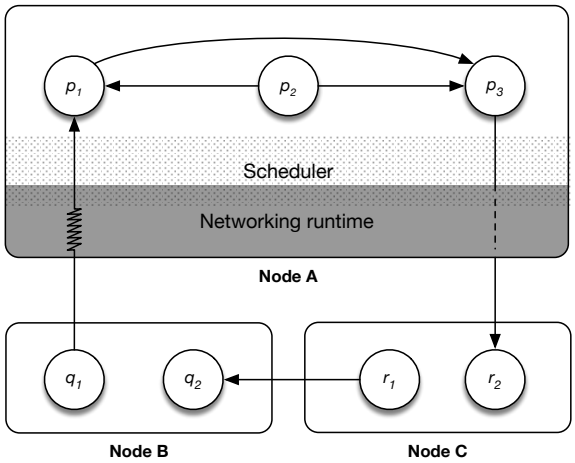

Figure 1: Nodes and processes in Troupe. The zigzagged line on the incoming message corresponds to message attenuation, the dashed line on the outgoing message corresponds to checking that the recipient is trusted to receive the message.

from received them is treated as public (i.e., confidentiality level $\bot$) and no confidential data can be sent to them.

In general, when node n_1 trusts node n_2 up to level ℓ it means:

1. only data labeled up to ℓ is sent from n_1 to n_2, and
2. data received from n_2 by n_1 is attenuated to be at most ℓ.

The first item prevents sending sensitive information to nodes that are not trusted to protect it. The second item weakens security labels of untrusted nodes.

We note that trust between nodes may be asymmetric, but is implicitly transitive. Trust should also not be conflated with integrity – in the current system we only focus on the confidentiality.

Figure 1 illustrates three Troupe nodes, each running a few processes. The arrows in the figure correspond to the messages between processes. Messages within each node are delivered to processes directly, whereas messages between nodes are subject to inspection by the networking runtime based on the trust levels.

2.2 TAG-BASED LABEL MODEL

The current version of Troupe uses a simple tag-based label model. Tags are abstract identifiers, e.g., `alice`, `bob`, `secret`, that specify confidentiality restrictions on data. A security level is a set of tags, e.g., ‘{`alice`, `bob`}‘, or ‘{`alice`, `bob`, `charlie`}‘. The more tags there are in the level the more restrictive is the data. For example, the level ‘{`alice`, `bob`}‘ is less restrictive than ‘{`alice`, `bob`, `charlie`}‘. The least restrictive level is the empty set level ‘{}‘. When two levels are ordered we write $\ell_1 \sqsubseteq \ell_2$ to say that level ℓ_2 is as restrictive as ℓ_1.

3 BASIC FEATURES

3.1 A MINIMAL TROUPE PROGRAM

We start with an example of a small Troupe program that returns number 42 as its result.

```
42
```

This program can be compiled, executed, and inspected for the return value using the following sequence of shell commands.

```
$ $TROUPE/bin/troupec program.trp -o out.js
$ $TROUPE/rt/troupe out.js --localonly
...
main thread finished with value: 42@{}%{}
```

The first command here invokes the Troupe compiler and specifies the generated output file to be `out/program.js`. The second command invokes the nodejs runtime on the generated file. Note `--localonly` flag that prevents the runtime from initializing the network layer – we discuss the network layer in Section 5. Troupe installation also includes a shell script `local.sh` that combines the above two tasks of compiling and running the programs.

Observe the format of the output value `42@{}%{}`. This value is *labeled* and consists of three parts: the base value 42, the security label of the value after the @ sign, and the security label of the type after the % sign. Here, `{}` that means the lowest security level.

3.2 OVERVIEW OF THE BASIC LANGUAGE

At the very core of Troupe is a dynamically typed functional programming language without mutable references. For example, a Fibonacci function in Troupe is written exactly as in SML sans type annotations.

```
(* basic_fib.trp *)
let fun fib x =
        if x > 2 then fib (x - 1) + fib (x - 2)
                 else 1
in fib 10
end
```

Running this program results in the output

```
main thread finished with value: 55@{}%{}
```

Note that Troupe is dynamically typed. For example, a program such as `1 + ()` results in a runtime type error

```
Runtime error in thread b279c491-4509-48e0-82fa-15f18de96003@{}%{}
   value ()@{} is not a number
```

3.3 TYPES AND VALUES

Troupe has the following types:

Unit Unit type with the unit value `()`.

Booleans Boolean type with literals `true` and `false`.

Number Number type.

String String type. String literals are created by placing text between double quotes, e.g., "hello, world".

Tuple Tuple aggregate, e.g., `("hello", (), true, 42)`.

List List aggregate, with empty list denoted as `[]`.

Function Function type. Function values are created either using `fn x => e` syntax or using the let-fun declarations (cf. Sec 3.5.4).

Process Id Process identifier (cf. Section 3.7).

Label Security label (cf. Section 4).

Authority Declassification capability (cf. Section 4.3).

Note that strings are also used for referring to node identities.

3.4 BASIC OPERATIONS

Troupe supports basic arithmetic and comparison operations on numbers. The comparison is also extended to strings. Aggregates support equality checks that is defined as point-wise equality of their elements. Boolean conjunction is `andalso`, and boolean disjunction is `orelse`. List cons operation is `::` and string concatenation is `^`.

3.5 THREAD-LOCAL CONTROL FLOW

Troupe uses SML-like syntax for thread-local operations. A reader familiar with SML may skip this subsection and proceed to Section 3.6.

3.5.1 If expressions

Conditionals have the form `if e0 then e1 else e2`, where `e0` is the guard of the conditional. The branch `e1` is chosen if `e0` evaluates to true; otherwise, the branch `e2` is chosen.

3.5.2 Case expressions

Case expressions have the form

```
case e of
   pat_1 => e_1
 | pat_2 => e_2
 |  ...
 | pat_n => e_n
```

Here, pat_1, ..., pat_n are the patterns that are matched against the expression e, and the e_1, ..., e_n are the expressions to evaluate. Patterns range over literals, including (), booleans, numbers, and string literals, as well as the aggregate constructors, and the wildcard pattern _.

3.5.3 Let expressions

Let expressions are used for binding names to values. The basic value binding uses the `let val` syntax. Several binding may appear in one block, and references to the earlier bindings in the block are allowed.

```
let val x = 20
    val y = x + 2
in x + y
end
```

3.5.4 Function declarations

Functions are declared using `let fun` syntax. Mutually recursive functions are delimited using the and keyword.

```
(* basic_evenodd.trp *)
let fun even x = if x = 0 then true else odd (x - 1)
    and odd x = if x = 0 then false else even (x - 1)
in even 7
end
```

Function application has the form `e0 e1`. Curried function declarations and partial application is supported, e.g.:

```
let fun add2 x y = x + y
    val inc = add2 1
in inc 10
end
```

3.6 LIBRARIES

Troupe has a minimal support for built-in libraries. Current libraries include a library `lists` for common list manipulations, library `declassifyutil` for declassification of aggregate data structures, and library `stdio` with convenient wrappers for standard input and output. To use a library, one needs to import it with an `import` statement at the top of the program.

```
(* basic_lib.trp *)
import lists
printWithLabels (map ( fn i => i + 1) [1,2,3])
```

3.7 CONCURRENCY

3.7.1 Spawning processes

New processes are created using `spawn` function that takes as its argument a unit-argument function. It return the process identifier of the new process, and starts the new process that from now on runs concurrently with the parent process.

```
(* basic_spawn.trp *)
import lists
let fun printwait x = let val _ = printWithLabels x in sleep 10 end
    fun foo () = map printwait [1,2,3]
    fun bar () = map printwait ["A", "B", "C"]
in  (spawn foo, spawn bar)
end
```

Execution of this program yields the following output:

```
main thread finished with value: (ae07ebcc-8a36-401e-b755-ff2393588c87@{}%{},
ed165a5d-acb0-4c6e-9843-29d9f263c6da@{}%{})@{}%{}
1@{}%{}
"A"@{}%{}
2@{}%{}
"B"@{}%{}
3@{}%{}
"C"@{}%{}
```

3.7.2 Sending and receiving messages

Process communication in Troupe happens via message passing. Every process has a designated mailbox, and other processes may send messages to that mailbox. Only the process that owns the mailbox may pick the messages from it.

To send a message to another process, we use function `send` that accepts as its argument a tuple of a process identifier and a value to send. To receive a message from the mailbox, we use function `receive` that accepts as its argument a list of handlers that select a message from the mailbox.

```
(* basic_receive.trp *)
let fun foo () =
      receive [hn x => printWithLabels ("foo␣received", x)]
    val p = spawn foo
in send (p, "hello")
end
```

Handlers In the example above, the expression `hn x => ...` is a *handler*. Handlers are used for selecting messages from the mailbox. The syntax for handlers is

$$\texttt{hn}\ pat\ \texttt{when}\ e_1\ \texttt{=>}\ e_2$$

Here, pat is the pattern to match, e_1 is the guard expression, and e_2 is the body of the handler. The guard part of the handler (`when` e_1) may be omitted. When several handlers are provided they are

used in the order they appear in the list. Finally, note that, on each invocation of `receive`, only one message is selected. Subsequent selection requires another call to `receive`. Finally, note that handlers are first class (see also the implementation note below).

Constraints on the guard expressions Evaluation of pattern and guard expressions is *sandboxed* – it may not invoke I/O operations, or spawn threads.

Implementation note Internally, handlers are desugared into a function with a case expression that returns a tuple such that the first element of the tuple indicates whether the message should be picked or not, and the second element is the body of the handler wrapped in a function, e.g., a value of the form `fn () =>` e_2.

3.7.3 Example: receive with a timeout

The following listing illustrates a program that implements receive with a timeout. In the code below, the primitive `sleep` suspends the execution of the thread for the specified number of milliseconds, while the primitive `mkuuid` creates a unique string. The use of the unique string avoids creating a confusion when multiple timeouts may be involved.

```
(* basic_timeout.trp *)
let fun timeout p r t = let val _ = sleep t
                        in send (p, r)
                        end
    val p = self ()
    val r = mkuuid ()
    val _ = spawn ( fn () => timeout p r 1000)
in receive [ hn ("MESSAGE", x) => print x
           , hn s when s = r => print "timeout"
           ]
end
```

3.7.4 Example: updateable service

The listing below presents an example of an updateable service – this is implemented by having a dedicated handler selecting on messages with the `"UPDATE"` string.

```
(* basic_updateableserver.trp *)
import timeout
let fun v_one n =
         receive [ hn ("REQUEST", senderid) =>
                         let val _ = send (senderid, n)
                         in v_one (n+1)
                         end
                 , hn ("UPDATE", newversion) => newversion n
                 ]

    val service = spawn (fn () => v_one 0)
    val _ = send (service, ("REQUEST", self()))
    val _ = receive [ hn x => print x]
```

```
    fun v_two n =
        receive [ hn ("REQUEST", senderid) =>
                        let val _ = send (senderid, n)
                         in v_two (n+1)
                        end
                , hn ("COMPUTE", senderid, f, x) =>
                        let val _ = send (senderid, f x)
                         in v_two (n+1)
                        end
                , hn ("UPDATE", newversion) => newversion n
                ]

    val _ = send (service, ("UPDATE", v_two))
    val _ = send (service, ("COMPUTE", self(), fn x => x * x, 42))
    val _ = receive [ hn x => print x]
in exitAfterTimeout authority 1000 0
        "force␣terminating␣the␣server␣example␣after␣1s"
end
```

The evaluation of this program results in the output

```
0@{}%{}
1764@{}%{}
main thread finished with value: ()@{}%{}
force terminating the server example after 1s
```

3.8 DEBUGGING CONCURRENT PROGRAMS

To help debugging concurrent programs, one can use a special primitive `_setProcessDebuggingName` that takes a string argument and uses that string when reporting errors in the process. By design, there is no mechanism for reading the process name other than causing an error; this prevents using process names to leak information.

4 INFORMATION FLOW CONTROL

This section presents the inner workings of Troupe's security monitor. The monitor is fail-stop at the granularity of individual processes: monitor violation in a process terminates that process but does not affect other processes or nodes.

The monitor is designed to enforce a variant of progress-sensitive noninterference with declassification. Progress-sensitive baseline is chosen because Troupe is a concurrent system that runs untrusted code, making it possible to amplify leaks via progress/termination, e.g., by designating a process per bit. In a dynamic system, such as Troupe, a progress leak may stem from several sources that includes divergence, blocking on input, or a runtime crash, such as evaluating the term `1 + ()`. All information flow violations result in termination of a process, unless the process is sandboxed.

4.1 PRIVILEGED OPERATIONS AND AUTHORITY

Troupe provides a set of privileged operations such as declassifications or process registration. All privileged operations require special *authority* values.

Authorities in Troupe are capabilities and are unforgeable. Operationally, authority is an encapsulated security level that we dub *efficacy* of an authority[2]. The higher the efficacy level the more powerful is the authority. System-wide privileged primitives, such as `register` in the echo-server example require the top authority, while declassification operations may use attenuated authority. Attenuation happens in one of the following two ways.

1. Programmatic attenuation takes place via a dedicated primitive `attenuate`. For example, expression `attenuate( authority, '{alice}')` returns authority value with efficacy `'{alice}'`. Programmatic attenuation helps programmers apply the principle of least privilege, for example, when passing authority to untrusted code that is allowed to perform some (but not all) declassifications.
2. Troupe runtime attenuates all levels and authority efficacies in remotely received data from ℓ to $\ell \sqcap \ell_{trust}$, where ℓ_{trust} is the trust level of the sending node.

In the beginning of program execution, Troupe runtime binds the top authority to the variable `authority` in the main program. This variable, however, is not in the scope of the code imported from libraries or received over the network. Such code needs to obtain authority explicitly.

4.2 MONITORING FOR INFORMATION FLOW

Labeled values Every value in Troupe is *deeply labeled* with confidentiality levels. The security level of a value specifies the confidentiality policy of the value. Troupe uses the syntax v`@'{`ℓ_{val}`}'%'{`ℓ_{type}`}'` to denote that the value v has security level ℓ_{val}, and the information about the type of this value is labeled at ℓ_{type}.

A labeled value can be created using Troupe's `raisedTo` primitive. Troupe's runtime propagates labels throughout the computation. This section presents examples of label propagation via explicit flows, implicit flows, and finally via termination and blocking.

Explicit flows The following example shows how to create labeled values and how explicit dependencies are propagated.

```
(* ifc_explicit.trp *)
let val x = 10 raisedTo '{alice}'
    val y = 20 raisedTo '{bob}'
in x + y
end
```

```
main thread finished with value: 30@{alice,bob}%{}
```

Relationship between type and value labels Observe how the result of this computation is labeled with the level `'{alice,bob}'`, but the type label is `'{}'`. Because information about the value is more precise than the information about the type, it holds that $\ell_{type} \sqsubseteq \ell_{val}$.

[2]In standard nomenclature this is simply "authority level". However, because authorities in Troupe are values with the corresponding value and type levels, we use a different term to avoid confusion.

Labels as values and label comparison Labels in Troupe are first-class:

```
(* ifc_labels.trp *)
import lists
map (fn (x,lev) => x raisedTo lev) [ (1, `{alice}`)
                                   , (2, `{bob}`)
                                   , (3, `{charlie}`) ]
```

```
>>> Main thread finished with value: [1@{alice}%{}, 2@{bob}%{},
3@{charlie}%{}]@{}%{}
```

Implicit flows Conditionals propagate the flows as well:

```
(* ifc_implicit1.trp *)
let val x = 10 raisedTo `{alice}`
in if x > 5 then 1 else 0
end
```

```
main thread finished with value: 1@{alice}%{alice}
```

Observe how the final value carries the dependency of the branch that has been chosen here. Because, of the dynamic nature of type checking, we also carry the information about the chosen branch into the type label.

Implicit flows and I/O The implicit information flows are further propagated between processes. In the mailbox, each message is additionally tagged with a label that corresponds to the blocking level of the sender at the time when the message was sent. Troupe supports filtering messages based on these tags. In the following example, `rcvp` is a form of receive statement that selects messages where the pc-tag set to the provided label argument.

```
(* ifc_implicit2.trp *)
let val x = 10 raisedTo `{secret}`
    val p = self()
    val _ = spawn ( fn () => if x > 0
                             then send (p, 1)
                             else send (p, 0) )
in rcvp (`{secret}`, [ hn x => x ])
end
```

```
main thread finished with value: 1@{secret}%{secret}
```

Termination and blocking flows In addition to tracking implicit flows via control flow, Troupe also tracks possible leaks via program termination or blocking. The sources of blocking are all synchronous operations such as receive statements or reading from standard input. Each thread has two runtime labels: the program counter label pc, and the blocking (termination) label $block$. It is always the case that $pc \sqsubseteq block$. To view the labels there is a helper internal function `debugpc()`.

```
(* ifc_debugpc.trp *)
let val x = 1 raisedTo `{secret}`
    val _ = debugpc()
```

```
    val x = if x > 2 then receive [] else ()
    val _ = debugpc()
in ()
end
```

Each call to debugpc prints out the information about the process identifier, the pc label, and the blocking label.

```
PID:2dfe86ae-6bf3-4bfc-8a0c-200230e0296c@{}%{}     PC:{}       BL:{}
PID:2dfe86ae-6bf3-4bfc-8a0c-200230e0296c@{}%{}     PC:{}       BL:{secret}
```

As one can see from the output above, the pc label after the conditional statement is lowered back to ‘{}‘, but the blocking label remains tainted with the label of the branch. The distinction between the two labels is helpful when declassifying the blocking label (cf. Section 4.3.3).

Implementation note Troupe implements information flow enforcement via inlining.

4.3 DECLASSIFICATION AND PROGRESS-SENSITIVITY

To relax the constraints imposed by the information flow control, Troupe offers a mechanism for *declassification*.

In Troupe, declassifying information requires a capability to declassify, known as the *declassification authority*. The authority carries a security level that is the upper bound on what information it can declassify. Given a value at level ℓ_{from}, an authority of level ℓ_{auth} permits a declassification to level ℓ_{to} if $\ell_{from} \sqsubseteq \ell_{to} \sqcup \ell_{auth}$.

4.3.1 Example with attenuation of authority

Authority can be attenuated using the attenuate primitive.

```
(* ifc_attenuate.trp *)
attenuate (authority, ‘{alice}‘)
```

```
>>> Main thread finished with value: !{alice}@{}%{}
```

4.3.2 Basic declassification

Troupe provides an explicit declassification command declassify that takes three arguments: the expression to declassify, the authority, and the target label. For example, a basic declassification looks like this.

```
(* ifc_declassify_atten.trp *)
let val x = 10 raisedTo ‘{alice}‘
 in declassify (x , authority, ‘{}‘)
end
```

When authority for declassification is not sufficient, Troupe returns an error message.

```
(* ifc_declassify_err.trp *)
let val authAlice = attenuate (authority, `{alice}`)
    val x = 1 raisedTo `{bob}`
in declassify (x, authAlice, `{}`)
end
```

```
Runtime error in thread b469cbef-16d3-4fe8-ac1e-bd21c7e8950d@{}%{}
>> Not enough authority for declassification
 | level of the data: {bob}
 | level of the authority: {alice}
 | target level of the declassification: {}
```

4.3.3 Declassification of the blocking level

The treatment of the blocking and termination labels is often restrictive in practice. To relax this, Troupe provides declassification of the blocking label. This is done using a variant of the `let` declaration that is called `let pini`. This statement requires the authority argument to declassify the blocking label after the sequence of the declarations (i.e., just before the `in` block). These declarations are accessible in the body of the let statement at permissive levels.

```
(* ifc_pini.trp *)
let val x = 10 raisedTo {alice}
    val y = 0
    val z =
        let pini authority
          val _ = if x > 1000 then receive [] else ()
          val z = y + 1
          val _ = debugpc ()
        in z
        end
    val _ = debugpc()
in z
end
```

```
PID:54bd148b-912e-4ae9-9787-aaf1166e9bc9@{}%{}     PC:{}       BL:{alice}
PID:54bd148b-912e-4ae9-9787-aaf1166e9bc9@{}%{}     PC:{}       BL:{}
>>> Main thread finished with value: 1@{}%{}
```

Observe that in the above program, the level of the final value `z` is not tainted by the blocking label of the high conditional.

Implementation note The expression `let pini e0` *decs* `in e1 end` is desugared by the compiler frontend into the form

```
let val tmp = pinipush e0
    decls
    val _ = pinipop tmp
in e1
end
```

Here, `tmp` is a fresh variable that is not part of the user program. Primitives `pinipush` and

pinipop dynamically scope the part of the execution for which the blocking label is declassified.

Implicit flows and type labels Type labels of values created in branches are also tainted by the pc-label. Operations that check the type label, e.g., arithmetics or pattern matching, use the information in the type label to appropriately taint the blocking level. the value.

```
(* ifc_type_labels.trp *)
let val x = 100 raisedTo {alice}
    val y = 200 raisedTo {bob}
    val z = 300 raisedTo {charlie}
    val a = let pini authority
                val a = if y > 10 then z else "not␣an␣integer"
            in a
            end
    val _ = printWithLabels a
    val _ = debugpc()
    val w = a + x
    val _ = printWithLabels w
in debugpc()
end
```

```
300@{charlie,bob}%{bob}
PID:2dfdc81f-1027-40e2-810c-11fadb2dd40f@{}%{}     PC:{}          BL:{}
400@{charlie,bob,alice}%{}
PID:2dfdc81f-1027-40e2-810c-11fadb2dd40f@{}%{}     PC:{}          BL:{bob}
```

4.4 INFORMATION FLOW CONTROL WITH I/O PRIMITIVES

This section may be omitted upon first read as it has a number of information flow subtleties that can be omitted when first starting to use the system.

4.4.1 Generalized receive and mailbox clearances

I/O operations such as send and receive introduce additional concerns w.r.t. information flow control. In particular, because mailbox acts as a mutable state, an extra care needs to be taken to control information flows through the mailbox structure.

Every message in a mailbox carries an extra label – the blocking level of the sender. We refer to this extra label as the *presence label*. Receiving a value furthermore propagates the taint of the blocking level from the sender to the receiver via the presence label. In order to constrain a receive operation from an accidental raise of the blocking level, Troupe provides a general receive primitive in the form `rcv(lev1, lev2, hns)`. Here, `lev1` and `lev2` indicate the to and from-levels of the presence labels on the messages, and `hns` is the list of handlers as before. The `receive` operation we introduced earlier is equivalent to `rcvp` at the level of the current pc label.

While such a form of general receive over an interval of levels is useful, it is also too powerful and needs to be further constrained. Consider the following snippet

```
let val _ = if secret then rcv ({}, {alice}, [hn x => x])
                      else ()
```

```
in rcv ({}, {}, [hn x => x])
end
```

Because the receive in the high branch is on an interval, the value of `x` may depend on the secret value. In general, one can encode an entire secret in the structure of the mailbox. To prevent such programs, Troupe provides a special notion of *mailbox clearance* that constrains receives on an interval. The mailbox clearance is a proxy for an authority. It can be raised using a dedicated command `raisembox(lev)` that returns a lowering capability and lowered with command `lowermbox(c, authority)`. There are a few constraints related to the mailbox clearance.

1. In order to receive on an interval (ℓ_1, ℓ_2) under pc with mailbox clearance ℓ_{clear} it must hold that $\ell_2 \sqcup pc \sqsubseteq \ell_1 \sqcup \ell_{clear}$. This constraint ensures that the mailbox clearance is sufficient for the interval receives. When mailbox clearance is $\bot$ – as it is in the beginning of the program – only point intervals of the form (ℓ, ℓ) where $pc \sqsubseteq \ell$ are allowed.
2. The pc-label of the program point where the mailbox clearance is raised affects the lower bound of the intervals. In particular, if the clearance is raised when the pc counter is pc_{raise}, the mailbox structure cannot be influenced by receives that are not as restrictive as pc_{raise}; in other words: $pc_{raise} \sqsubseteq pc \sqcap \ell_1$, where ℓ_1 is the lower bound of the interval receive.
3. If the process mailbox clearance is raised in a branch, it must be lowered back before reaching the join point of the branch.

5 NETWORKING

5.1 NETWORK IDENTITY

Troupe's runtime connects to a distributed p2p system. Each instance of the runtime is associated with a network node that uses a unique network identifier. The identifier information is typically stored in files, and is provided as arguments to the runtime at startup.

```
$TROUPE/rt/troupe out.js --id=<path_to_id_file>
```

If no arguments are provided, the runtime generates a fresh identifier at startup.

5.1.1 Registering and looking up processes

Node identifiers can be used to register and look processes up. Consider a simple echo service:

```
let val _ = register ("echo", self(), authority)
    fun loop () =
        let val _ = receive [ hn ("ECHO", x, sender)
                                => send (sender, ("REPLY", x))
                            , hn _ => () ]
        in loop ()
        end
in loop ()
end
```

The `register` primitive is used to bound a process to a name. Because this is a system-wide operation, it requires the top authority argument. If we know the node identity and the name at which a process is bound, we can find the process at that node, using the `whereis` primitive:

```
let val echo =
        whereis ( "QmNRwNZACPciLS14cZFApwrCcAdbRAXYgztea9m5XwRe4z"
                , "echo")
    val _ = send (echo, ("ECHO", "Hello", self()))
in receive [ hn  x => print x]
end
```

The complete echo example – including the code above and the scripts for generating the identifiers is available at `$TROUPE/examples/network/echo`.

5.1.2 Aliases

Aliases is a mechanism that allows us to avoid having hardcoded node identifiers in the source code. The alias mechanism operates in the way of the traditional Unix hosts files. To use this mechanism, we need to provide the path to a special alias file that contains a mapping between alias strings and network identifiers. In the text of the program, an alias string must start with character "@".

```
$TROUPE/rt/troupe out.js --aliases=<path_to_aliases_file>
```

5.2 NODE TRUST LEVELS

By default, all nodes in Troupe are mutually distrusting. This means that information sent and received to other nodes is considered to be at level ‘{}‘. We can increase trust in certain nodes by passing a trust map file that include the identity of the nodes and their maximum trust level.

5.3 REMOTE SPAWNING

If we know of a node identifier, we can spawn a thread on that node. Remote spawning is disabled by default, and needs the flag `--rspawn=true` to be enabled. To spawn a process on a remote node, the `spawn` takes a tuple of arguments, where the first parameter is the string corresponding to the node identifier of the remote machine.

5.4 INFORMATION FLOW MONITORING AND ATTENUATION

When information at level ℓ_{data} is sent to a remote node with trust level ℓ_{trust}, the runtime performs the check $\ell_{data} \sqsubseteq \ell_{trust}$ to ensure that no sensitive information flows to a that can violate confidentiality. Because we have no way of enforcing the information flow on the remote node, this also means that trust relationship between nodes is transitive.

When receiving information from a remote node at level ℓ_{trust}, data labeled at level ℓ receives the actual level $\ell_{trust} \sqcap \ell$. This ensures that the runtime is not accidentally tainted by nodes that have low (or none at all) trust.

Authority values that are transferred across the nodes are subject to the same constraints and attenuation.

A LANGUAGE REFERENCE

A.1 BUILT-IN EXPRESSIONS

A.1.1 `adv`

Description Simulate sending a value to adversary at level $\{\}$. This function is introduced as a pedagogical convenience as as it removes the necessity to set up a network process when explaining explicit and implicit information flows.

Arguments A value to pass to the adversary.

Returns Unit.

Failure behavior Crashes the current process if the provided value and the blocking label are more restrictive than bottom.

Example usage `adv` (`42 raisedTo {alice}`).

A.1.2 `attenuate`

Description Returns the attenuated authority

Arguments A value of authority type.

Returns A value of authority type

Example usage `attenuate(authority, '{alice}')` Note that the above example will generate a runtime error.

A.1.3 `authority`

Description Return the authority argument implicitly provided to the top-level function of the main program. The accessibility of this argument follows the standard lexical scoping rules. In particular, expressions received over the network carry over the (potentially attenuated) authority of their originating nodes.

Arguments None

A.1.4 `declassify`

Description Declassifies an expression.

Arguments A triple of the form $(expr, \texttt{authority}, \ell)$, where $expr$ is the expression to be declassified, `authority` is the authority to use for declassification, and ℓ is the target level of declassification.

Returns The original value declassified to the target level if there is sufficient authority.

Failure behavior Crashes if the provided authority is insufficient for the declassification.

Example usage
```
let val x = 42 raisedTo {alice}
in print (declassify (x, authority, `{}`)
end
```

A.1.5 `exit`

Description Exits the Troupe runtime.

Arguments A tuple (`authority`, `exitCode`).

Returns Nothing

Example usage `exit(authority, 0)`

A.1.6 `getTime`

Description Obtains current Unix timestmap

Arguments Unit.

Returns Number.

Example usage `getTime()`

A.1.7 `inputLine`

Description Reads a line from the console.

Arguments Unit.

Returns String value.

Blocking behavior Synchronous. Observe that the blocking label is raised to top after this operation, because the user providing the input is assumed to operate at level top. To prevent this from happening, the blocking label can be declassified using the `let pini` constructs. See also library functions `inputLineWithPini` and `inputLineWithPini` from `stdio` library.

A.1.8 `lowermbox`

Description Lowers the clearance of the current process' mailbox.

Arguments A tuple of the raise capability and authority

Returns Unit.

Failure behavior Fails if the type of the argument is invalid (dynamic type checking). Fails if the authority is insufficient for this lowering, or the provided capability does not match the stack scoping discipline.

A.1.9 `mkuuid`

Description Generates a random string.

Arguments Unit.

Returns A newly generated string.

Example usage `print (mkuuid ())`

A.1.10 `node`

Description Returns node identifier of a process.

Arguments Process identifier.

Returns String containing a node identifier.

A.1.11 `pinipop`

Description Pop an authority value from the *pini* stack, and declassify the current blocking level.

Arguments A string generated by `pinipush`

Returns Unit.

Failure behavior Crashes if the popped authority is insufficient for the declassification of the blocking level, or if the provided capability does not match to the last capability generated by the `pinipush`.

Example usage `pinipop ()`

A.1.12 `pinipush`

Description Pushes authority value onto the *pini* stack.

Arguments A value of type authority.

Returns A string capability to be passed as the argument to `pinipop`

Example usage `pinipush (authority)`

A.1.13 `pinipushto`

Description Pushes authority value onto the *pini* stack, with explicit blocking level.

Arguments A tuple: a value of type authority, and a level.

Returns A string capability to be passed as the argument to `pinipop`

Failure behavior Crashes if the current blocking level does not flow to the level argument.

Example usage `pinipushto (authority, {bob})`

A.1.14 `print`

Description Prints a value on the console omitting its security types.

Arguments A value of any type.

Returns Unit.

Example usage `print "Hello, world"`

A.1.15 `printWithLabels`

Description Prints a value on the console including its security types.

Arguments A value of any type.

Returns Unit.

Example usage `printWithLabels "Hello, world"`

A.1.16 `raiseTrust`

Description Dynamically raise the trust level of a node.

Arguments A triple of a node identifier, root-level authority, and the intended trust level.

Returns Unit.

Failure behavior . Fails if the argument type is invalid. Fails if the authority argument is not top. Fails if the blocking level is not $\bot$.

Example usage `raiseTrust("@alicescomputer", authority, {alice})`

Note that the top-level authority is required because this is a privileged operation with system-wide consequences.

A.1.17 `raisembox`

Description Raises the clearance of the current process' mailbox.

Arguments A security level

Returns A capability for lowering the mailbox level back to the previous value.

Failure behavior Fails if the type of the argument is invalid (dynamic type checking).

A.1.18 `random`

Description Generates a random number between 0 (inclusive) and 1.

Arguments Unit.

Returns Number.

Example usage `random()`

A.1.19 `receive`

Description Picks a message from the mailbox.

Arguments A list of *handler functions* (cf. Section 3.7.2).

Returns The value returned by the body of the matching handler.

A.1.20 `register`

Description Registers the process under a name.

Arguments A tuple of the form $(str, pid, authority)$, where str is the name under which the process is to be registered, pid is the process identifier, and $authority$ is the top authority value.

Returns Unit.

Blocking behavior Synchronous.

Failure behavior Crashes if the provided authority is not top, or if the blocking level is not $\bot$.

Example usage `register ("auctionServer", self(), authority)`

Note that this is a privileged operation with system-wide consequences.

A.1.21 `rcv`

Description Picks a message from the mailbox with pre-filtering the levels of the messages.

Arguments A triple of the form $(hs, \ell_{lo}, \ell_{hi})$ where hs is the list of handlers, ℓ_{lo} is the lower bound on the *sender-level* of the messages, and ℓ_{hi} is the upper bound on the *sender-level* of the messages in the mailbox.

Returns The value returned by the body of the matching handler.

A.1.22 `sandbox`

Description Execute a function in a sandbox for a fixed duration of time.

Arguments A tuple of the form (t, f), where t is the timeout duration in milliseconds, and f is the function of the form `fn`$() =>$ `e` that is executed in the sandbox. A sandboxed process is heavily restricted: it cannot perform any I/O operations or spawn new threads.

Returns A tuple of the form (ok, val) where ok is either true or false, with true meaning that the sandbox execution has completed successfully, and false meaning that the sandbox has exhibited a crash or timeout. When the evaluation succeeds, value val carries the return value of the sandboxed function; it is a unit value otherwise.

Blocking behavior Synchronous. The execution of the process always takes at least duration t, even if the sandboxed process successfully finishes before that.

Failure behavior This function always succeeds. All internal errors are suppressed.

Example usage `sandbox (1000, fn () => 1 + () )`

Implementation notes Troupe's current prototype does not yet implement this with high granularity.

A.1.23 `self`

Description Returns the current process identifier.

Arguments Unit.

Returns The value of process identifier type.

Example usage `print (self ())`

A.1.24 `send`

Description Sends a message to a process.

Arguments A tuple of the form (pid, v) where the pid is the process identifier of the recipient process, and v is the value to send.

Returns Unit.

Failure behavior Crashes the current process if the node hosting the recipient process is untrusted to receive v. This crashing behavior is exhibited only if the process is hosted remotely. Local processes are implicitly fully-trusted because the enforcing runtime is the same.

Blocking behavior Asynchronous.

A.1.25 `_setProcessDebuggingName`

Description Sets the process name that is used in debugging.

Argument A string value.

Returns Unit.

Failure behavior Fails if the type of the argument is invalid (dynamic type checking).

A.1.26 `sleep`

Description Suspends the execution of the current thread for a specified duration of time.

Arguments One argument of type number that specifies the sleep time in milliseconds.

Returns Unit.

Example usage `sleep 100`

A.1.27 `spawn`

Description Spawns a new process locally or remotely.

Arguments Either a tuple of the form (`nodeid`, `f`) of a node identifier (as a string) and a function or one argument of the type function. Note that the function must be of the form `fn`() => `e`, i.e., it must take just one unit argument.

Returns The process identifier of the newly spawned process.

Blocking behavior Blocks until the new process is created.

Example usage `print (self (), spawn(fn () => print (self())))`

A.1.28 `raisedTo`

Description Raises the level of a value. Observe that this expression uses the infix syntax.

Arguments An expression of any type and a level.

Returns The same expression with its level raised to the provided level.

Example usage `42 raisedTo {alice}`

A.1.29 `whereis`

Description Looks up a registered process on a remote node.

Arguments A tuple of the form $(nodeid, name)$ where $nodeid$ is the string representing the p2p-identifier of the node, and $name$ is the name of the process registered at that node.

Returns The process identifier of the node on the remote machine.

Blocking behavior Synchronous.

Failure behavior Crashes if the program counter level and the level of the $name$ do not flow to the trust level of the node.

Example usage
```
let val p = whereis
                ( "QmeuSjy8RbeUHpqtDGj2B66fPHxRowuoecZnaVkvhTS7tb"
                , "auctionServer")
    in send (p, ("BID", 42))
    end
```

B USEFUL STANDARD LIBRARIES

Current installation of Troupe has a minimal set of standard libraries. We briefly describe the most useful ones here.

B.1 LIBRARY `DECLASSIFYUTIL`

This library contains some useful functions for *deep declassification* of tuples and declassifying both an expression and blocking level at the same time.

- `declassify_with_block`$(\texttt{v}, \texttt{auth}, \texttt{lev})$ takes three arguments – the value to declassify, the authority, and the target level – just like in normal declassification – and declassifies both the blocking label and the value.
- $(\texttt{declassifydeep}(\texttt{v}, \texttt{auth}, \texttt{lev})$ is like abov, but additionally further declassifies nested lists and tuples up to size nine.

B.2 LIBRARY `LISTS`

This library contains standard list functions, such as `map`, `mapi`, `foldi`, `range`, `range`, `reverse`, `lookup`, `elem`, `length`, `append`, and `partition`.

C INSTALLATION AND CONFIGURATION

C.1 INSTALLATION

See the `README` file provided with the installation.

C.2 CONFIGURING NETWORK IDENTIFIERS

Create a network identifier (together with the public/private key pair) for your node.

```
$ curl lbs-troupe.troupe-lang.org/mkid -o my-identifier.json
```

The `"id"` section of the JSON file is your identifier. The other parts correspond to the private and public keys – technically, the identifier is the hash of the public key.

C.2.1 Testing your network identifier (optional)

At this step, you can pass the generated JSON file as an argument to Troupe runtime, using the `--id` flag. See the echo example provided with the installation.