Towards Secure Distributed Computations

III ReSeCo Workshop

Felipe Zipitría

Grupo de Seguridad
Instituto de Computación
Facultad de Ingeniería - UdelaR

November 25th, 2008

Outline

Outline

This talk will be about

- Distributed computations
- Distributed programming methodology
- Proof checking
- Proof Carrying Results
- Security

Outline

Distributed computations

- Grids - Volunteer computing
  - SETI@HOME
  - Distributed.NET (RC5)
  - PrimeGrid (Mersenne #45 and #46 $<$ 2 weeks!)

SETI Detected problems

Incorrect results were returned!

- from overclockers
- from modified algorithms

Verification technique

Result checking

Distributed computations

- Grids - Volunteer computing
  - SETI@HOME
  - Distributed.NET (RC5)
  - PrimeGrid (Mersenne #45 and #46 $<$ 2 weeks!)

SETI Detected problems

Incorrect results were returned!

- from overclockers
- from modified algorithms

Verification technique

Result checking

Distributed computations

- Grids - Volunteer computing
  - SETI@HOME
  - Distributed.NET (RC5)
  - PrimeGrid (Mersenne #45 and #46 $<$ 2 weeks!)

SETI Detected problems

Incorrect results were returned!

- from overclockers
- from modified algorithms

Verification technique

Result checking

Distributed programming methodology

- Computational framework
- Components
- Modularisation
- Abstraction

Outline

Distributed programming languages

Common/Desirable properties

- A language with a module system which permits us ?to model ADTs
- Simplified communication of arbitrary values between different processes
- Safety along the distributed infrastructure/runtime

Safety

Relevant for our distributed infrastructure

We will focus on:

Type-safety

(Progress + preservation) a.k.a. Soundness

Abstraction-safety

Semantics and type system of the programming language guarantee abstraction protection

Safety

Relevant for our distributed infrastructure

We will focus on:

Type-safety

(Progress + preservation) a.k.a. Soundness

Abstraction-safety

Semantics and type system of the programming language guarantee abstraction protection

Simplified communication

Definitions

Marshalling The process of gathering data and transforming it into a standard format before it is transmitted over a network or saved to a permanent format.

Unmarshalling Reverse process, which transforms data from standard format back to its original form.

Introducing Acute

- Research language: INRIA Rocquencourt + University of Cambridge
- ML-like core, with extensions to support distributed development
- Provides safe and robust mechanisms to develop and execute separately-built programmes

Acute features

- Allows cooperating programmes to send and receive values through (untyped) communication channels
- Supports distributed computation of values providing (un)marshalling procedures
- Primitives for type-safe (un)marshalling

$$e ::= \quad \ldots \mid \text{marshal } e_1 \; e_2 : T \mid \text{unmarshal } e : T \mid \ldots$$

Particular Acute features

Type-safe (un)marshalling

- Types are hashed to be used by the type checker
- Dynamic type-check at unmarshal time
- Type equality is defined simply by equality on hashes
- Guarantees both type-safety and abstraction-safety

Acute modularisation

Signatures (Interfaces)

```
module Prime:
  sig
    type t
    val start: t
    val get: t -> int
    val next: t -> t
  end
```

Structures (Modules)

```
=
  struct
    type t = int
    let start = seed
    let get x = x
    let next x = some_alg x
  end
```

Extending on Acute type equality

Hashing

- A notion of type equality that makes sense is needed across the entire distributed system
- Type *Prime*.t is compiled to *h*.t, where the hash *h* is (roughly)

$h \equiv hash$ (

```
module Prime:
sig
 type t
 val start:t
 val next: t -> t
 val get: t -> int
end
=
struct
type t = int
 let start = seed
 let get x = x
 let next x =
some_alg x
end
```

)

Acute respects abstractions

Example A

prog a = send(marshal 5:int)

prog b = module Prime =

```
module Prime           =
: sig                  struct
 val init: t           type t = int
 val get: t -> int      let init = (a seed)
 val next: t -> t       let get x = x
end                     let next x = some_alg(x)
                       end
```

print_int(Prime.get (unmarshal (receive():Prime.t))

This computation should fail

machineA[*prog a*] | machineB[*prog b*]

Acute respects abstractions

Example A

```
prog a =  send(marshal 5:int)
prog b =  module Prime              =
          : sig                     struct
           val init: t              type t = int
           val get: t -> int         let init = (a seed)
           val next: t -> t          let get x = x
          end                        let next x = some_alg(x)
                                    end
print_int(Prime.get (unmarshal (receive():Prime.t))
```

This computation should fail

machineA[*prog a*] | machineB[*prog b*]

Outline

Active adversary

- An entity who tampers with data

Who to trust...

What happens if some attacker steals our type hash?

- Typecheck is made by the sender, before marshal

In the context of active adversaries in our network

$prog\ a' =$ send ($raw\_marshal$ {8,hash(module Prime)})
$prog\ b =$ (same as example A)

Invariant of ADT Prime broken!

print_int (Prime.get (unmarshal (receive ()):Prime.t))

- There is no typechecking of values of abstract data types (other than hash equality)

Who to trust...

What happens if some attacker steals our type hash?

- Typecheck is made by the sender, before marshal

In the context of active adversaries in our network

$prog\ a' =$ send ($raw\_marshal$ {8,hash(module Prime)})
$prog\ b =$ (same as example A)

Invariant of ADT Prime broken!

print_int (Prime.get (unmarshal (receive ()):Prime.t))

- There is no typechecking of values of abstract data types (other than hash equality)

Who to trust...

What happens if some attacker steals our type hash?

- Typecheck is made by the sender, before marshal

In the context of active adversaries in our network

prog a' = send (*raw_marshal* {8,hash(module Prime)})
prog b = (same as example A)

Invariant of ADT Prime broken!

print_int (Prime.get (unmarshal (receive ()):Prime.t))

- There is no typechecking of values of abstract data types (other than hash equality)

What happens in Acute?

- It is type-safe in a trusted setting
- Works well if we can typecheck our values correctly
- What happens with values of abstract data types? Representation, for checking, is not available

Summary

Context

Distributed programming languages
Abstract data types
Communications are made with primitives (un)marshal

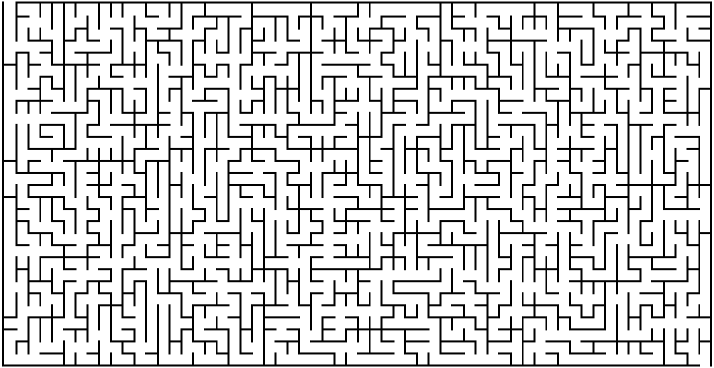
Objective

Definition and experimentation with mechanisms that permit us to strengthen abstraction-safety properties

Outline

Maze example

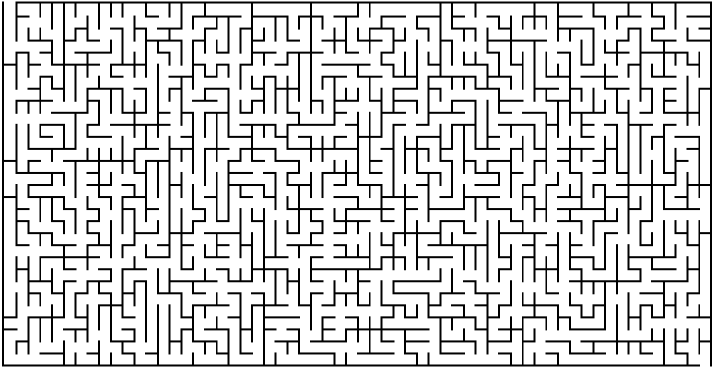
Can you find the way out?



- Can you do it faster?

Maze example

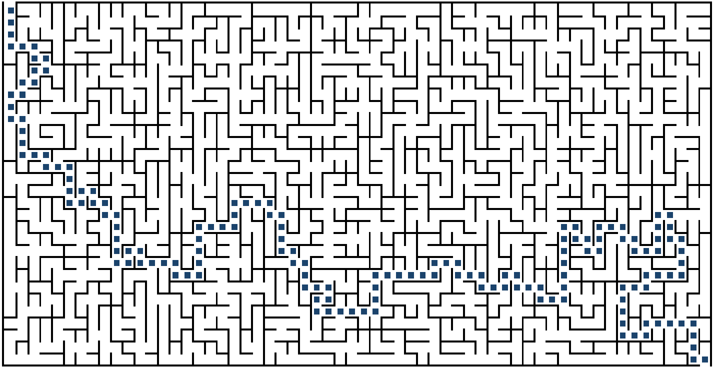
Can you find the way out?



- Can you do it faster?

Approach

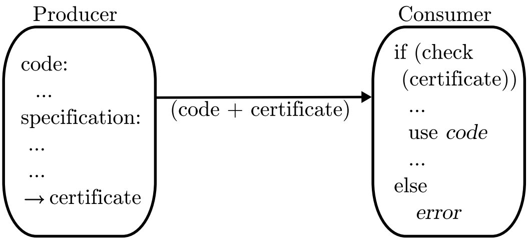
Maze solved!



- What if we follow the blue dots? $\Longrightarrow$ trivial to find the way out...

Proof Carrying Code

Example

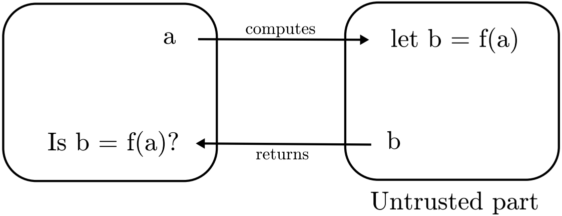


- Code sent to a remote consumer has a certificate
- Certificate is a formal safety proof
- Shows that the code complies with certain specification of safety rules

Proof Carrying Result

Approach & Scheme

- Reuses concepts from PCC
- Based on verification
- Distributed computation among untrusted hosts

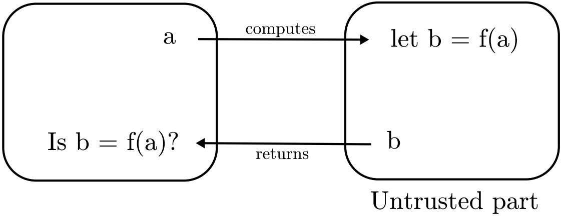


- We need a way to check that $f(a) = b$ … but without computing $f(a)$
- A certificate

Proof Carrying Result

Approach & Scheme

- Reuses concepts from PCC
- Based on verification
- Distributed computation among untrusted hosts



- We need a way to check that $f(a) = b$ ... but without computing $f(a)$
- A certificate

Proof Carrying Result

More Formally

Some Definitions

$f \in A \to B, a \in A$

$f(a)$ is delegated to an untrusted party

We must have a function

$check_f \in AxB \to bool \mid \forall (a,b) \in AxB, check_f(a,b) = true \Rightarrow b = f(a)$

- Every function f?

Proof Carrying Result

More Formally

Some Definitions

$f \in A \rightarrow B, a \in A$

$f(a)$ is delegated to an untrusted party

We must have a function

$check_f \in AxB \rightarrow bool \mid \forall (a,b) \in AxB, check_f(a,b) = true \Rightarrow b = f(a)$

- Every function f?

Proof Carrying Result

More Formally

Some Definitions

$f \in A \rightarrow B, a \in A$

$f(a)$ is delegated to an untrusted party

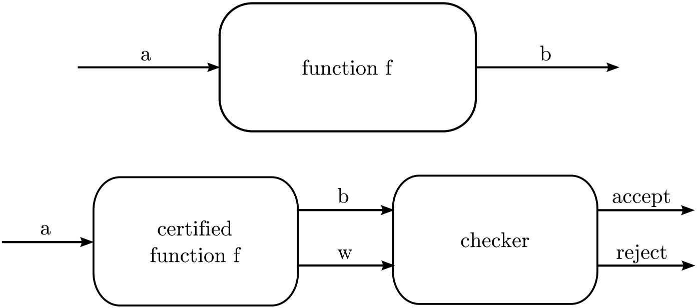
We must have a function

$check_f \in AxB \rightarrow bool \mid \forall (a,b) \in AxB, check_f(a,b) = true \Rightarrow b = f(a)$

- Every function f?

Outline

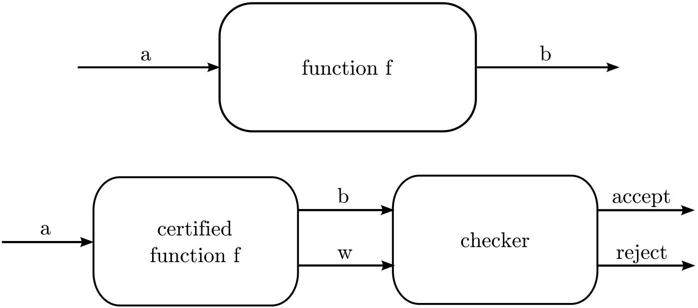
Approach



Fact

Not every algorithm is certifying

Approach



Fact

Not every algorithm is certifying

Simple examples

$$GCD(x,y) = d \text{ where } d \mid x \wedge d \mid y \wedge (\forall d', d' \mid x \wedge d' \mid y \Rightarrow d \mid d')$$

Certified

$\Longrightarrow ExtendedGCD(x,y) = (u,v,d)\, where\, d \,|\, x \wedge d \,|\, y \wedge d = ux + vy$

Sorting a list L

Certified

$\Longrightarrow$ Sorting a List L, and giving its sort order

Adding PCR

- In order to add this technique, we must define an infrastructure for it
- Certified result communication

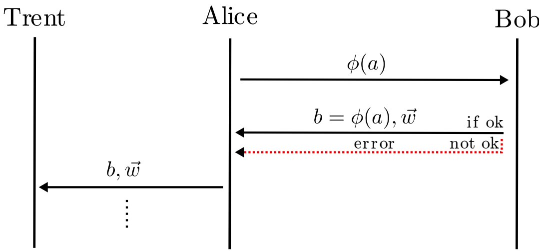
Outline

PCR Protocol

Alice a consumer of remote computations

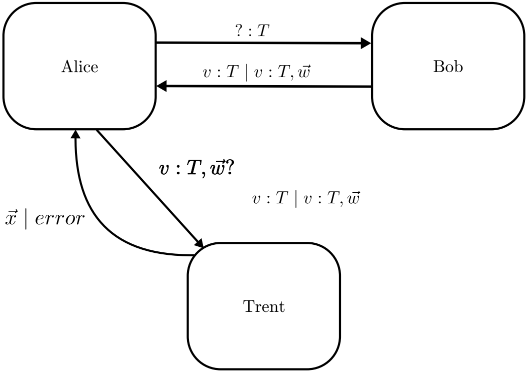
Bob an untrusted producer

Trent a trusted arbitrator



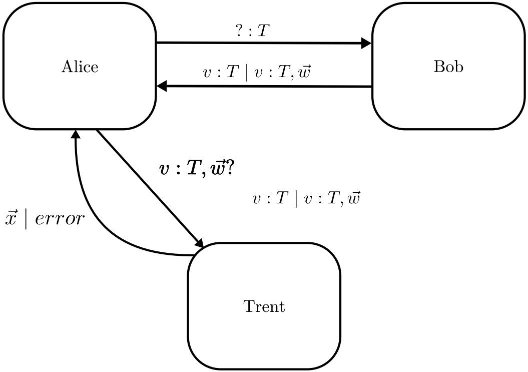
We defined a protocol for doing PCR computations

Infrastructure model



- Simplifying, *Alice* $\equiv$ *Trent*
- We will focus on $v : T, \vec{w}$

Infrastructure model



- Simplifying, *Alice* $\equiv$ *Trent*
- We will focus on $v : T, \vec{w}$

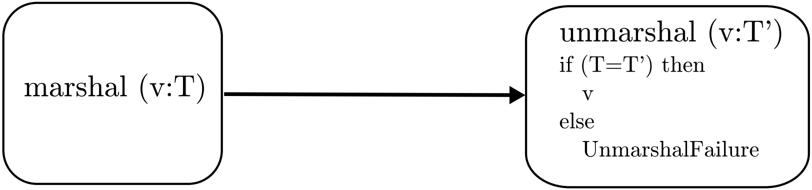
Outline

Current (un)marshal primitives

Syntax and semantics

$marshal\ e_1\ e_2 : T$

$unmarshal\ e : T'$



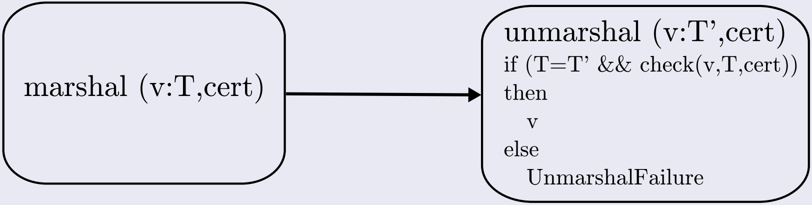
New (un)marshal primitives

Syntax and semantics

Extended with certificate

$marshal\ e_1\ e_2 : T < certificate >$

$unmarshal\ e : T' < certificate >$



How was the language modified?

- Core modifications:
  - Lexer
  - Parser
  - Abstract Syntax Tree
- Contact point with the PCR infrastructure
  - A way to check certificates

What is a certificate?

Our certificate

Assertion for a property
Proof of that assertion
Another assertion...

In the implementation

type assertion = string
type proof = string
type certificate = (assertion * proof) list

Extension of the Acute language

Objective

Increase abstraction-safety

When?

$\Longrightarrow$ It will be added at unmarshal time

Means

Checking for a certificate

Extension of the Acute language

Objective

Increase abstraction-safety

When?

$\Longrightarrow$ It will be added at unmarshal time

Means

Checking for a certificate

Extension of the Acute language

Objective

Increase abstraction-safety

When?

$\Longrightarrow$ It will be added at unmarshal time

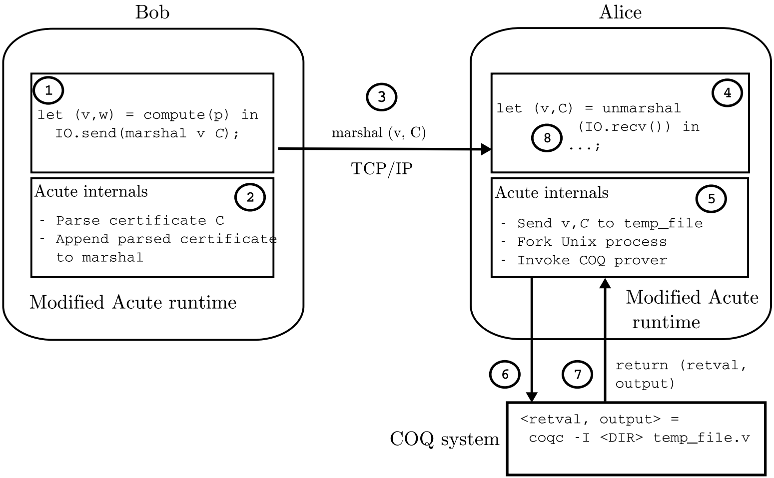
Means

Checking for a certificate

Checking certificates

- Proof verification process of the result certification was performed using COQ
  - theoretic support
  - large user community
- COQ usage
  - the value, its type and certificate are written to a file
  - call the COQ compiler on this file

Full proof sequence



Some Details

To be taken into account

- COQ
  - <DIR> is not arbitrary: it must be part of the *Trusted Computing Base*
    - it is a list of filesystem directories that have COQ properties
  - Unwanted or problematic commands are filtered out (*e.g.* *Axiom*, *Parameter*)
- The certificate must prove required properties
- Authenticity of results

Case study: Certified prime number generation

Pocklington's criteria

Given a natural number $n > 1$, a witness a, and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:

$$p_1 \ldots p_k \textit{ are prime numbers} \tag{1}$$

$$(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) \mid (n-1) \tag{2}$$

$$a^{n-1} = 1 (\bmod n) \tag{3}$$

$$\forall i \in \{1, \ldots, k\} \gcd(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{4}$$

$$p_1^{\alpha_1} \ldots p_k^{\alpha_k} > \sqrt{n} \tag{5}$$

Pocklington's criteria

Certificate

The numbers $a, p_1, \alpha_1, \ldots, p_k, \alpha_k$ constitute a Pocklington certificate.

- Used by the CoqPrime project to certify primes

In the previous example

prog a′ = send (*raw_marshal* {8,hash(module Prime), <cert?>})
prog b = (same as example A)

- A certificate cannot be constructed for that value of abstract type $\surd$

Contributions

- An infrastructure has been defined and implemented for supporting the technique of proof carrying results,
- the Acute distributed programming language has been extended, with a mechanism that permits the exchange of values of abstract types in a certified way, and
- for performing the verification of the results, this infrastructure has been connected with COQ.

Conclusions

- We have defined and implemented an infrastructure for doing proof carrying results
- The infrastructure is independent of the language
- Working with a proof checker is a good way of delegating the checking process
- Proof Carrying Results is a new approach
    - Its progress depends on the development of certifying algorithms
- Extending the chosen language was a complex task

Future work

- Distribute certificate checking/generation
- Integrate the infrastructure defined with other distributed languages
- Only perform the certificate check if the type of the received value is abstract
- COQ proof checker: have a proof “server”

Towards a distributed certifying infrastructure

Distributing work

- A certificate is a vector $\vec{w} = (w_1, w_2, \ldots, w_n)$, where each of the w_i is an assertion
- And for all of these assertions, we have

$$\forall i \begin{cases} w_1 & \mathit{proved} \Rightarrow \mathit{Prop}(p_1, w_1) \\ w_2 & \mathit{has\ not\ been\ proved} \\ \ldots & \ldots \\ w_j & \mathit{proved} \Rightarrow \mathit{Prop}(p_j, w_j) \\ \ldots & \ldots \\ w_n & \mathit{has\ not\ been\ proved} \end{cases}$$

Distribution

We can distribute verification between hosts

- Proof obligations generator?

Towards a distributed certifying infrastructure

Distributing work

- A certificate is a vector $\vec{w} = (w_1, w_2, \ldots, w_n)$, where each of the w_i is an assertion
- And for all of these assertions, we have

$$\forall i \begin{cases} w_1 & proved \Rightarrow Prop(p_1, w_1) \\ w_2 & has\ not\ been\ proved \\ \ldots & \ldots \\ w_j & proved \Rightarrow Prop(p_j, w_j) \\ \ldots & \ldots \\ w_n & has\ not\ been\ proved \end{cases}$$

Distribution

We can distribute verification between hosts

- Proof obligations generator?

Questions?