GRACKLE

Automated, Verified Message Encoding and Decoding

Matt Schwennesen – 29 September 2025

Outline

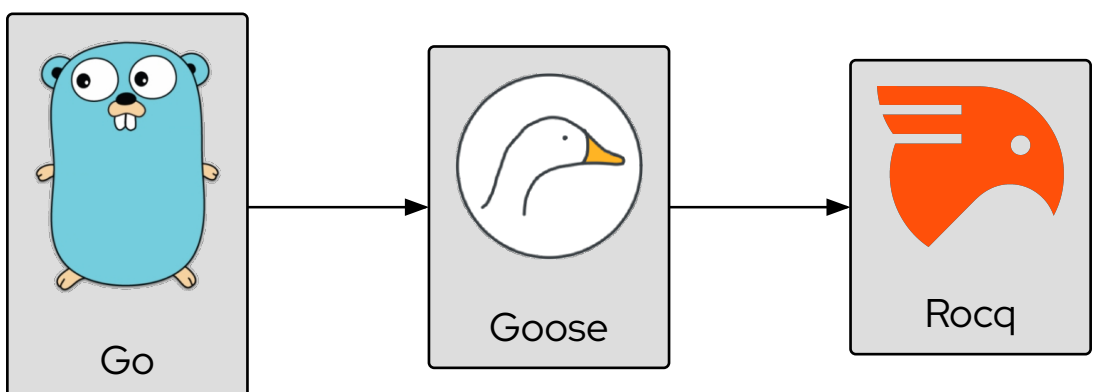
1. Motivation
2. Grackle Overview
3. Messages
4. Enums
5. Map Literals
6. Evaluation
7. Future Work
8. Questions?

MOTIVATION

Why Verify Distributed Systems?

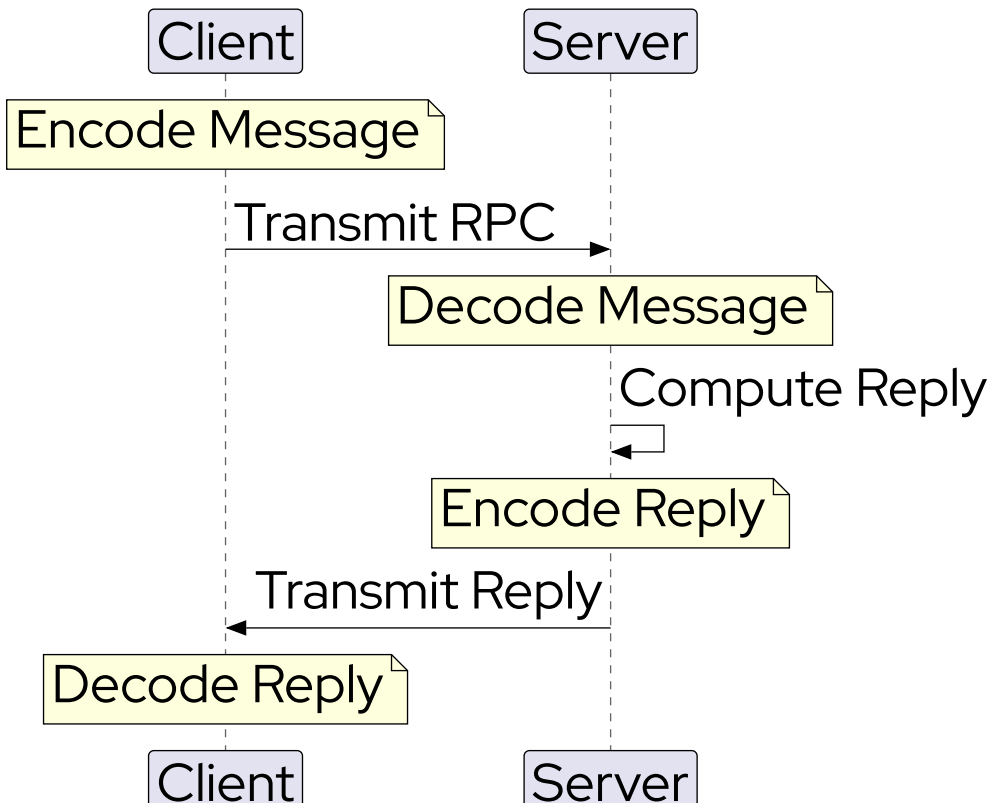
- Concurrency introduces new classes of bugs.
- Testing cannot prove the absence of bugs.
- Formal verification can provide assurances.

Goose & Perennial



- Separating Conjunction: $*$
- Affine Logic: Resources can be used *at most once*.

Grackle's Motivation



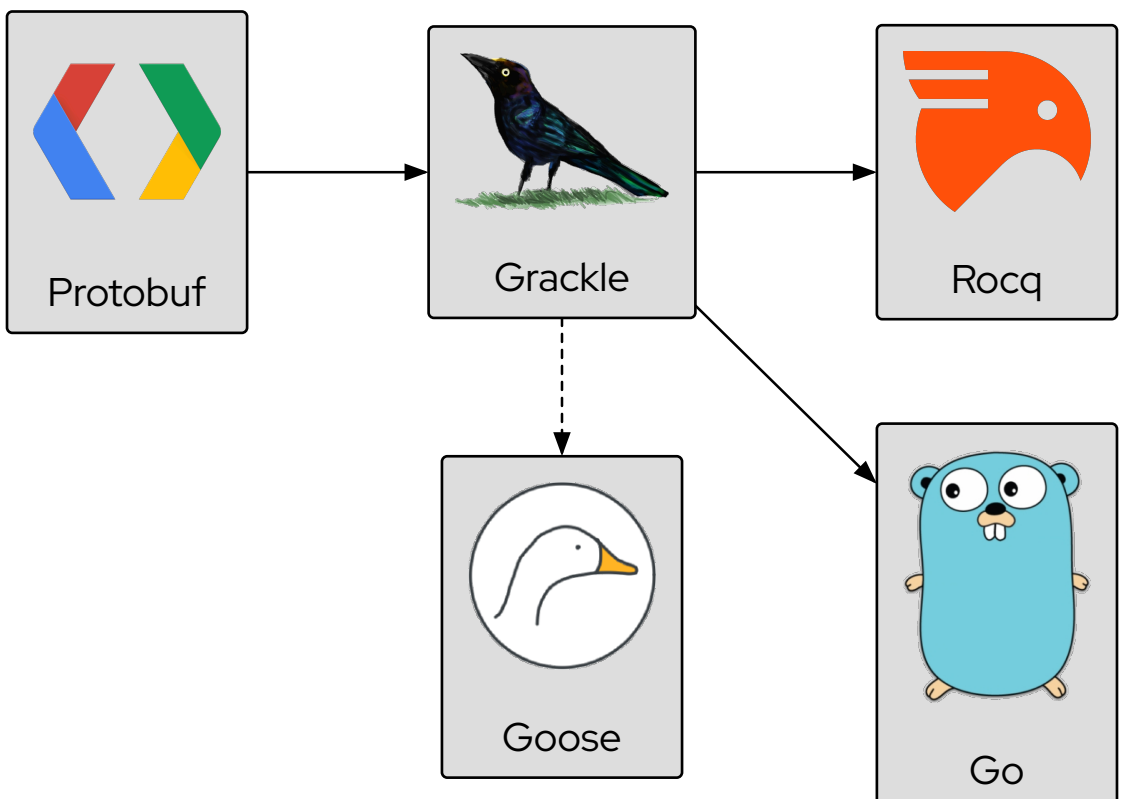
Grackle's Motivation

Existing encoding and decoding functions serialize each field in a `struct` in order.

Proofs over these functions are *repetitive boilerplate* that is **amiable to automation**.

GRACKLE OVERVIEW

Inputs & Outputs



Alternatives

What are the other options?

- Write encoding and decoding proofs manually
- Verify a global encoding system once and for all

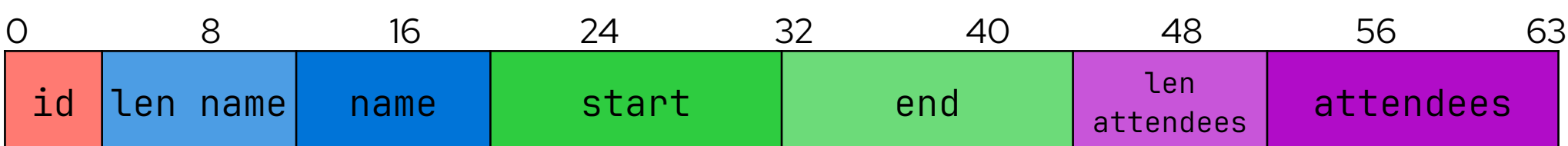
Grackle's Internals

- Written in `go`.
- Uses the `text/template` package.
- Unverified but proof instrumented.
- Uses a custom `go` API and wire format.

Wire Format

Protocol Buffer

```
1 message event {
2   uint32 id = 1;
3   string name = 2;
4   timestamp startTime = 3;
5   timestamp endTime = 4;
6   repeated user attendees = 5;
7 }
```



Supported Features

There are two different types of structures supported by Grackle, *messages* and *enums*

For messages

- Scalar types: unsigned integers, Booleans, strings, bytes[1]
- Nested messages
- Repeated fields
- Local imports

[1]Modeled somewhere between a scalar and repeated field, `repeated bytes` not supported.

MESSAGES

Message Representation

Protocol Buffer

```
message event {
  uint32 id = 1;
  string name = 2;
  timestamp startTime = 3;
  timestamp endTime = 4;
  repeated user attendees = 5;
}
```

Go

```
type S struct {
  Id        uint32
  Name      string
  StartTime timestamp_gk.S
  EndTime   timestamp_gk.S
  Attendees []user_gk.S
}
```

Message Representation

Protocol Buffer

```
message event {
  uint32 id = 1;
  string name = 2;
  timestamp startTime = 3;
  timestamp endTime = 4;
  repeated user attendees = 5;
}
```

Rocq

```
Record C := mkC {
  id' :  u32;
  name' :  go_string;
  startTime' :  TimeStamp_gk.C;
  endTime' :  TimeStamp_gk.C;
  attendees' : list User_gk.C;
}.
```

Simple Message Optimization

A message is *simple* if it

- Doesn't contain a `enum` field
- Doesn't contain a `repeated` field
- Doesn't contain a `bytes` field
- Doesn't contain a message which isn't simple

Simple messages allow the proof engineer to not track goose level and proof level `struct` contents.

Marshal & Unmarshal

Grackle generated `go` code uses a *stateless* API

```go
1   func Marshal (prefix []byte, e S) []byte {
…                                          …
12  }
```

- Must provide a prefix.

```go
1   func Unmarshal (s []byte) (S, []byte) {
…                                          …
18  }
```

- Returns a suffix.

Ownership in Perennial

- `own`: Links the specification level `struct` to the goose level `struct`.

```
1 Definition own (args_v: event_gk.S.t) (args_c: C) (dq: dfrac): iProp
```
Rocq

  - Each field matches
  - Strings are less than 2^{64} bytes
  - Lists are less than 2^{64} elements
- `has_encoding`: Links the specification level `struct` to the encoding.

```
1 Definition has_encoding (encoded: list u8) (args: C) : Prop
```
Rocq

  - Encoding is built of encoding of each element
  - Strings are less than 2^{64} bytes
  - Lists are less than 2^{64} elements

Encoding & Decoding Theorems

Rocq

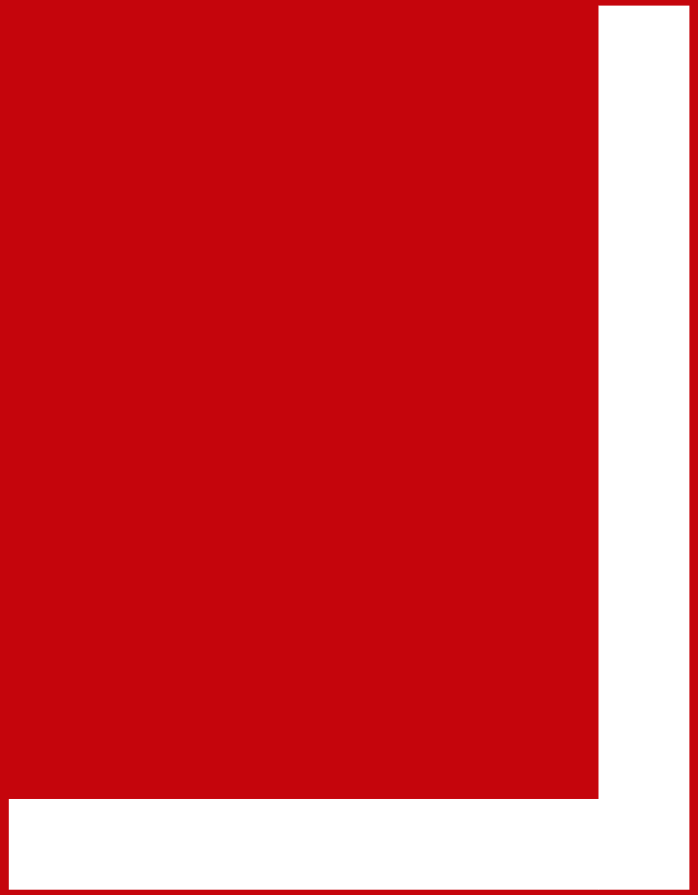
```
1   Lemma wp_Encode (args__t : event_gk.S.t) (args__c : C) ... :
2     {{{   is_pkg_init event_gk *
3           own args__t args__c dq *
…                                          …
6     }}}
7       @! event_gk.Marshal #pre_sl #args__t
8     {{{   enc enc_sl, RET #enc_sl;
9           ⌜ has_encoding enc args__c ⌝ *
10          own args__t args__c dq *
11          own_slice enc_sl (DfracOwn 1) (prefix ++ enc) *
…                                          …
13    }}}.
```

Encoding & Decoding Theorems

Rocq

```
1  Lemma wp_Decode (enc : list u8) (enc_sl : slice.t) (args__c : C)
   (suffix : list u8) (dq : dfrac):
2    {{{   is_pkg_init event_gk ∗
3          ⌜ has_encoding enc args__c ⌝ ∗
4          own_slice enc_sl dq (enc ++ suffix)
5    }}}
6      @! event_gk.Unmarshal #enc_sl
7    {{{   args__t suff_sl, RET (#args__t, #suff_sl);
8          own args__t args__c (DfracOwn 1) ∗
9          own_slice suff_sl dq suffix
10   }}}.
```

ENUMS



Enum Representation

Protocol Buffer

```
1  enum error {
2    eOk = 0;
3    eEndOfFile = 1;
4    eUnknown = 2;
5  }
```

Go

```
1  type E uint32
2
3  const (
4    EOk        E = 0
5    EEndOfFile E = 1
6    EUnknown   E = 2
7  )
```

Enum Representation

Protocol Buffer

```
enum error {
  eOk = 0;
  eEndOfFile = 1;
  eUnknown = 2;
}
```

Rocq

```
Inductive I :=
| eOk
| eEndOfFile
| eUnknown.

Definition to_tag i : w32 :=
  match i with
  | eOk => W32 0
  | eEndOfFile => W32 1
  | eUnknown => W32 2
  end.
```

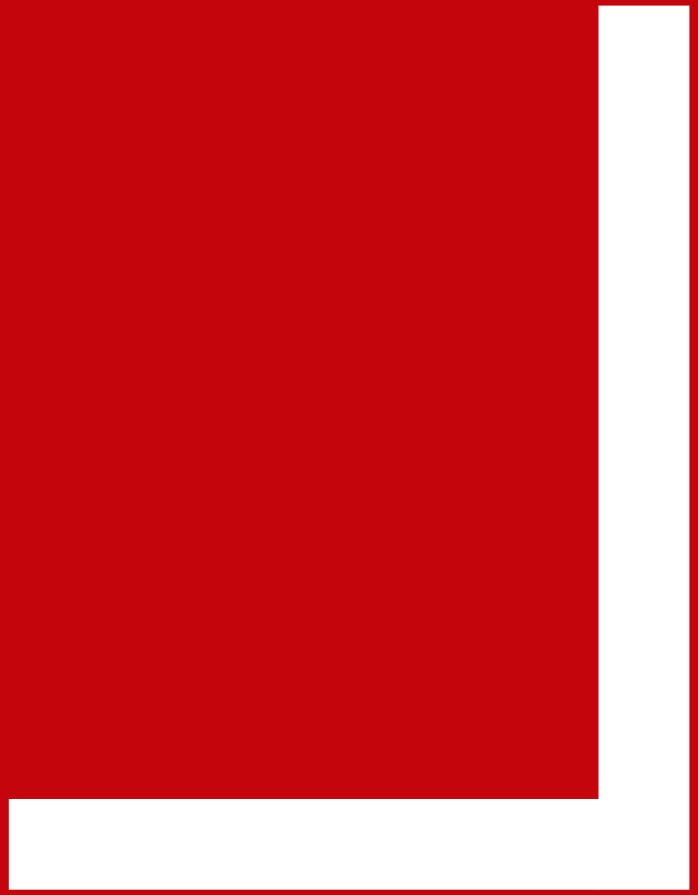
Enums: Open or Closed?

The `go` code uses an *open* enum while rocq uses a *closed* enum.

- Protobuf states that all enums should be open[2].
- Closed enums are easier to reason about in rocq.

[2] https://protobuf.dev/programming-guides/enum/

MAP LITERALS



Enum Maps

```go
var Name = map[uint32]string{
  0: "eOk",
  1: "eEndOfFile",
  2: "eUnknown",
}
```

```go
var Value = map[string]uint32{
  "eOk":        0,
  "eEndOfFile": 1,
  "eUnknown":   2,
}
```

When using enums, Grackle provides helper maps as global variables.

Improving Map Literal Support

Perennial already had support for maps, but not map literals

- Define new goose lang construct syntax and semantics.
- Prove `wp_map_literal`

EVALUATION

Internal Testing

Grackle's internal test suite contains a representative sample of protobuf definitions.

Proto Lines	Messages	Enums	Go Lines	Proof Lines	Spec Lines
86	10	1	526	658	754

Gokv

Gokv is a collection of distributed key-value stores often used to test new Perennial features.

Proto Lines	Messages	Enums	Go Lines	Proof Lines	Spec Lines
381	54	7	2301	2370	3498

Limitations

- Missing proper `map` support (1 instance)
- Cannot directly use `repeated bytes` (1 instance)

FUTURE WORK

Future Work

- Signed Integers
- Support `maps`
- Support `repeated bytes`
- Protobuf Compatibility
- `oneof` fields
- Usage in more projects

QUESTIONS?