

Towards Verified Time Balancing of Security Protocols

Donovan Crichton

January 2019

Motivation

- ▶ ASD manually verifies vendor code with containing cryptographic processes.
- ▶ Formal Methods: A mathematically based approach to the specification and verification of software.
- ▶ Can we reduce some of the resources ASD spends on manual verification by replacing with automatic verification?
- ▶ Can we also further research into secure communications protocols?
- ▶ A case study.

Formally Verifying a Time Balanced Security Protocol

- ▶ Attackers can gain information from message timing.
- ▶ Can we model a time-invariant protocol?
- ▶ A naive approach considers all operations have the same running time.
- ▶ Can we ensure that assumptions on this model hold for the implementation?

The ZRTP Protocol

- ▶ Initially started with ZRTP.
- ▶ ZRTP is complex and makes many decisions.
- ▶ Simplified version that contains just enough detail to allow us to attempt to prove some interesting things!

The Simplified Protocol

- ▶ Commit messages that contain SHA-256 hashes
- ▶ Diffie-Hellman key exchange contains modulo arithmetic.
- ▶ How can we formally guarantee the timing of operations?
- ▶ Lets cover some background before diving in!

Approach

- ▶ We can use the notion of propositions as types.
- ▶ Functional programming languages that support dependent types can act as theorem provers under a higher order constructive logic.
- ▶ We can model this protocol in such a language, Idris.
- ▶ We can express proofs about properties of the protocol.

Quick Background 1 - Currying

All functions treated as taking a single argument.

$$f : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

$$f = +$$

Applying an argument to a multi argument function returns the rest of the function! (Arrow associates to the right)

$$f(2) : \mathbb{N} \rightarrow \mathbb{N}$$

$$f(2) = 2+$$

Finally all arguments are applied.

$$f(2, 3) : \mathbb{N}$$

$$f(2, 3) = 2 + 3 = 5$$

Quick Background 2 - Idris Syntax and Values as Types

Building a vector type in Idris:

```
data Vec : Nat -> Type -> Type where
  Nil    : Vec 0 a
  (::)    : (x : a) -> Vec n a -> Vec (n + 1) a
```

We can parameterise types over values to capture invariants in the model.

```
append : Vec n a -> Vec m a -> Vec (n + m) a
append Nil ys = ys
append (x :: xs) ys = x :: append xs ys
```


Quick Background 3 - Propositions as Types

Under the assumptions of referential transparency and totality.

Logic Term	Logic Symbol	Idris Symbol	Idris Type
Implication	$p \Rightarrow q$	$p \rightarrow q$	Function
Conjunction	$p \wedge q$	(p, q)	Pair / Tuple
Disjunction	$p \vee q$	$\text{Either } p \text{ } q$	Tagged Union
Negation	$\neg p$	$p \rightarrow \text{Void}$	Void Type
IFF/Eq	$p \equiv q, p \iff q$	$(p \rightarrow q, q \rightarrow p)$	Pair Arrows
Universal	$\forall x. P \ x$	$p \rightarrow \text{Type}$	Π Type
Existential	$\exists x. P \ x$	$(x ** P \ x)$	Σ Type
		$p = q$	Type Equality

Correctness by construction

- ▶ The protocol implementations can be quite complex, possibly giving rise to equally complex proof obligations.
- ▶ We'd like to reduce this burden somehow!
- ▶ We can design a language that is just big enough to compute what we like, but restrictive enough to capture the invariants we care about.

Building a type of Prg n

Statement	Continuation	Result	Description
Halt	-	Prg 1	Terminate
AssC	Prg k	Prg ($k + 1$)	Asn constant.
AssV	Prg k	Prg ($k + 1$)	Asn variable.
UnOp	Prg k	Prg ($k + 1$)	Asn result of unary op.
BinOP	Prg k	Prg ($k + 1$)	Asn result of binary op.
Do	Prg k	Prg ($m * n + k$)	Run Prg m, n times.
Cond	Prg k	Prg ($n + k$)	Branch on Prg n.
Skip	Prg k	Prg ($k + 1$)	Do Nothing.

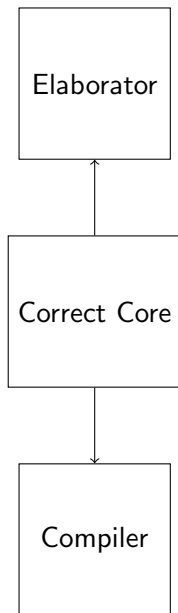
Conditionals require both branches to be Prg n. Ensuring That all branches of the program are correct by construction.

What about more expressive time parameters?

```
f : Value -> Value -> Nat
f = --some complex function to calculate the number of
    --computation steps for each input to mod.

data Prg : Nat -> Type where
  Mod : {k : Nat} -> (cont : Prg k) -> (x : Value) ->
    (y : Value) -> Prg ((f x y) + k)
```

Elaboration and Compilation of a Correct Core



- ▶ The small, correct core language can be elaborated to a more full-featured language.
- ▶ The size of the core language makes the burden of proofs much lighter.
- ▶ The compiler can map the core language expressions down to something more “real world” (e.g C, assembler).

Contributions

- ▶ Formal description of a simplified protocol.
- ▶ Prg: A small language parameterised over computational time
- ▶ Some small proofs of Prg correctness.

Further Work

- ▶ Implement the simplified protocol in Prg.
- ▶ Modulo arithmetic cases.
- ▶ Relax some of the (many) assumptions.
- ▶ Investigate elaboration and compilation with regard to invariants.