Modeling High-Level Public-Key Cryptography in Coq

Andres Erbsen

May 17, 2015

1 Introduction

Cryptographic protocols are a critical piece of the common Internet infrastructure. Yet the ways of gaining assurance about the correctness of cryptographic protocols are unsatisfying at best. The gold standard is a computational proof of security by reduction from a well-known computational problem widely believed to be intractable. However, this kind of analysis is requires substantial mathematical rigor and persistence, especially if done quantitatively as appropriate for real systems. Furthermore, the classical proofs are not modular: a proof that protocol P respects some security property will need to be redone for every combination of extensions to P. As a practical example of this, base TLS 1.2 is secure both when used with either channel binding authentication or session resumption, but fails to provide any meaningful security if the two extensions are enabled together[1].

As an alternative to computational reduction proofs, Dolev-Yao-style symbolic have been used. Instead of assuming compromise and deriving a contradiction, properties are proved by showing that no operation in the postulated list of what the adversary can do could violate the desired property. Proofs in this style are much less technically sophisticated, but the setup is error-prone: for example, omitting the fact that textbook RSA encryption and multiplication are commutative from the list of adversary's capabilities can allow for "verification" of an exploitable protocol.

We seek to build a formal model of high-level cryptographic operations similar to those provided by NaCl[2] that would allow to prove secrecy and authentication properties of protocols that use them, for example SIGMA[3]. All guarantees would be with respect to a "reasonable" number of messages and computational limitations of the adversary *as defined by the specifications of the primitives*; our

[1]https://www.secure-resumption.com/#channelbindings
[2]http://nacl.cr.yp.to/
[3]http://webee.technion.ac.il/~hugo/sigma-pdf.pdf

model will not involve any quantitative reasoning. Similarly, we assume that the specifications of the primitives capture all information flow from their inputs to their outputs – proving the lack of side channels in an implementation is not a goal. This paper presents a model of public-key encryption and its application to reasoning about execution traces of cryptographic protocols; we also discuss using the core message passing concurrency library from Verdi to reason about all possible executions of protocols.

1.1 The Subject of Verification

This work is motivated by the observation that many protocols tend to achieve their secrecy and authentication properties constructively in a particular sense explored in this section. Each step (encryption, checking of an authenticator, selectively revealing information) creates a divide between desirable and undesirable scenarios. This is an observation about protocol design, not analysis: in fact, the most of the protocols covered here have been analyzed using both computational and symbolic models. To show an example of this design pattern, let's consider the CurveCP protocol (table 1) which provides a TLS-like encrypted channel. The protocol (1) establishes an ephemeral Diffie-Hellman secret between an externally authenticated long-term public key of the server and a randomly generated public key of the client; (2) then uses an embedded encrypted "vouch" packet to say that the public key the server saw indeed belongs to some desired client. The first step enables clients to tell the good server from an impostor, and the second lets the server distinguish the specific client from others. These intended results can be translated to the language of state invariants as follows: (1) for all clients in the state right after receiving the server's first message, the value that the client has stored as "the servers' ephemeral public key" is the public counterpart of some secret generated by the server and never revealed to anybody else and (2) all servers in the state right after receiving a vouch subpacket have the value stored as "client's ephemeral public key" be the public counterpart of a value generated and kept secret by the client who the server has listed under "client identity" for the same session.

Table 1: CurveCP connection initialization from `curvecp.org`.

Hello packet: `(C',0,Box[0'](C'->S))` where `C'` is the client's short-term public key and `S` is the server's long-term public key and `0` is zero-padding and `0'` is zero-padding	
	Cookie packet **(1)**: `(Box[S',K](S->C'))` where `S'` is the server's short-term public key and `K` is a cookie
Initiate packet with Vouch subpacket **(2)**: `(C',K,Box[C,V,N,...](C'->S'))` where C is the client's long-term public key and `V=Box[C'](C->S)` and `N` is the server's domain name and `...` is a message	

There are three things we consider noteworthy about the reasoning in the previous paragraph. First, it is **boring** in the sense that the three ways of stating the two points read almost the same to somebody with experience in working with cryptography. The protocol itself is fairly straightforwardly derived from the possible scenarios it must distinguish, and the invariants say the same thing while being slightly more careful about quantifiers. Second, it is actually **non-trivial**. The history section of the extended version of Hugo Krawczyk's SIGMA paper[4] tells a story of how designing a similar protocol using low-level primitives lead to exploitable misconceptions in designs used in IPSec. Figure 1 even challenges the reader to tell bogus versions from good ones, and even though the each variant is adequately described using a simple three-line arrows-and-messages diagram, the correct answer is not obvious. Last but not the least, the reasoning presented about CurveCP is **modular**: in the actual protocol, the server does send its ephemeral public key over the network (encrypted using a symmetric key that is never revealed), but arguing that this optimization does not jeopardize safety is independent from the proof of the authenticity property. We believe having this kind of flexibility in a safe manner is hugely important for more realistic use case.

To stress the importance of design flexibility, let's look at some examples. The general bidirectional vouching structure advocated for in the SIGMA paper is applicable in a wide variety of scenarios, but the specific message flow is subject to a numerous other constraints related to relationship privacy, denial-of-service-attack resilience, latency, and the underlying transport. For this reason, three extremely similar variants of the SIGMA protocol are to be hand-verified

[4]http://webee.technion.ac.il/~hugo/sigma-pdf.pdf

separately in a computational model, which to the best of our knowledge has not been done. An even better example of this issue is the Axolotl family of protocols[5] used in Pond and TextSecure: while conceptually equivalent to CurveCP, a long list of additional constraints (for example, forward secrecy, protection from compromise in the past, both of these for content and metadata) has forced the protocol to continuously perform Diffie-Hellman key exchanges and hash ratchet steps to provide the same guarantees in a wider range of scenarios. Indeed, the extra complexity incurs a cost: as demonstrated in[6], a stripped down version of Axolotl does exhibit some behavior that is sufficiently counter-intuitive to be called an attack by a well-intentioned interpretation by a third party (even though the authors of the protocol argue that the property which the reported attack breaks cannot in practice hold in any protocol[7]).

One way of looking at this work is that we are creating a framework for showing that particular refinements of a general protocol template provide the desired properties in spite of changes required by other design constraints. This specialization enables us to simplify our reasoning significantly: we axiomatize how a system can possibly keep a value secret in an almost syntactic manner inspired by taint tracking, and reduce proving secrecy in a single execution of a protocol to proving a non-taintedness of the execution trace. Reasoning about all possible executions is enabled by the message passing concurrency library from Verdi. Some protocols can not be verified in this manner because they use lower-level primitives for which we do not have a taint-tracking-style model. For example, we do not expect this approach to work for realistic OTR, SSHv2.0 or SSL (even though verification of an idealized version may succeed). We consider this limitation to be an analog of a better-known principle about verification of software implementations: it is easier to build a verified system bottom-up than to prove correctness of an independently written codebase.

2 Preservation of Secrecy

To introduce or mechanism for reasoning about secrecy, let's first consider a simpler problem: how to show that a network message does not leak a secret key. In our model, the network message is produced by evaluating an expression with the current state and possibly some stimuli as its inputs. One approach would be to show that the value being sent to the network is the same for all possible values of the secret key. Proving this would involve applying the fact `f_equal` (applying the same function to the same inputs produces identical outputs) recursively over the entirety of the expression tree. This is equivalent to showing that the secret key does not appear in the expression by showing that it does not appear in any of the subexpressions. To simplify extending this kind

[5]https://github.com/trevp/axolotl/wiki

[6]https://eprint.iacr.org/2014/904.pdf

[7]From: Moxie Marlinspike moxie@thoughtcrime.org; To: messaging@moderncrypto.org; Date: Sat, 01 Nov 2014 00:24:17 -0700

of reasoning to encryption, we choose the second route: we axiomatize that the value v is indistinguishable from other possible values of the same type given the expression e if it is indistinguishable given each argument to the function at the root of e. This requires us to represent the expression tree as an explicit Gallina object. However, as we are only interested in the structure of the expression tree and not the details of its behavior, we leave the types and functions (including constants) as opaque Gallina objects. The following is a simplified version of our expression language that only models traditional taint propagation.

```
funcType : list Type -> Type -> Type (* argTs -> retT -> funcT *)
call : funcType ?args ?T -> hlist id ?args -> ?T
Inductive exp : Type -> Type :=
| External' : forall {T: Type} {argts} (f:@funcType argts T) (args:hlist exp argts), exp T.

Inductive IND : forall {V T:Type}, V -> Exp T -> Prop :=
| INDArgs' : forall {V T} (v:V) argts f args,
  (forall W (e:Exp W) (pf_m:member W argts), IND v (hget args pf_m))
  -> IND v (@External T argts f args).
```

Writing down the indistinguishability relation is a non-trivial technical task: the standard advice for keeping proofs simple would point towards using a `list_all` predicate to assert the indistinguishability from the arguments, yet this is not possible because the strict positivity requirement of CIC does not let us pass a closure producing `IND` to some other function inside the declaration of `IND`. Instead, we state the inductive predicate using a nested `forall` quantifying over all expressions that appear in the arguments list and stating the independence property about them. As this definition does not directly allow proofs by case analysis on a known argument list, we prove a separate lemma showing that the definition we actually want implies the one that fits the requirements of CIC. This proof took considerable effort and required using the `dependent destruction` tactic from the `Program` library to perform case analysis on proof arguments. The lemma is formally stated as follows:

```
forall {V T} (v:V) argts f args
    hAll (fun t (e:Exp t) => IND v e) args
      -> IND v (@External T argts f args).
```

2.1 Modeling Public Key Encryption

We chose to model the `crypto_box` encryption scheme from `NaCl`[8] because it provides a useful and intuitive security guarantee with very modest requirements on how it is used. We further simplify our analysis by ignoring the concerns

[8]https://cryptojedi.org/papers/coolnacl-20111201.pdf

of nonce generation: `crypto_box` is secure when used with a random nonce[9], and while some protocols do use a counter instead, the issue of making sure that nonces do not repeat is not the part of security we wish to model. We also set aside the concern of the length of the ciphertext revealing information about the message: it as well can be dealt with separately. We would like to stress that the formalization did not force our hand here; we are optimistic that modeling message length leakage, sender public key leakage, and the nonce management requirements is simply a matter of adding more preconditions to the rule that allows inferring the secrecy of an encrypted message. We also assume for simplicity that all errors are handled by ignoring the message whose handling caused the error because this is the case for all protocols we think could be verified using our approach.

However, with these details taken care of, the public key encryption security guarantee[10] becomes fairly simple: if a key that was generated randomly is used to encrypt a message, the chance that an attacker will be able to distinguish that message from another possible message is negligible as long as the attacker's knowledge of the secret key is limited: it is safe to reveal the public key and encryption and decryption results. This does not directly fit into the taint tracking system, but only because the taint propagation rules depend on what may already have leaked. We work around this by adding an additional argument, the context containing all current and previous messages. We also need to axiomatize that our randomly generated keys are independent of each other and user input. Our final secrecy rules are shown in the following code snippet, along with a proof that the smallest use case, encrypting a message using a fresh key and revealing ciphertext, does not reveal the message.[11].

```
Section IND.
  Context `{params : CryptoParams}.

  Variable C : Type.
  Variable ctx : Exp C.
  Inductive IND : forall {V T:Type}, V -> Exp T -> Prop :=
  (* Normal taint tracking: *)
  | INDArgs' : forall {V T} (v:V) argts f args,
         (forall W (e:Exp W) (pf_m:member W argts), IND v (hget args pf_m))
         -> IND v (@External T argts f args)
  (* Random number generation: *)
  | INDBoxNewNeq : forall idx1 idx2, idx1 <> idx2 ->
         IND (Unop box_new idx1) (Unop box_new idx2)
  | INDBoxNewSym : forall {T} (x:T) idx,
         IND (Unop box_new idx) (Symbolic x)
  (* Public-Key encryption: *)
```

[9]http://nacl.cr.yp.to/box.html

[10]https://eprint.iacr.org/2001/079

[11]`unop`, `binop`, and `terop` lift Gallina functions to our embedded expression language

```
  | INDBoxKeygen : forall idx,
        IND (Unop box_new idx) (Unop box_keygen (Unop box_new idx))
  | INDBoxMessage : forall {V M} (v:V) (m:Exp M) idx pk,
        IND (Unop box_new idx) ctx ->
        IND v (Terop (@box _ M) (Unop box_new idx) pk m)
  | INDBoxOpen : forall {V M} (v:V) (m:Exp M) sk pk sk' pk',
        IND v m ->
        IND v (Terop (@box_open _ M) sk pk' (Terop (@box _ M) sk' pk m))
  (* Branching (sel flag t f := if flag then t else f): *)
  | INDSel : forall {T} (v:T) (l r:Exp T) (br:Exp bool),
        IND v (sel (eval br) l r) -> IND v br -> IND v (Terop sel br l r)
  .

Section ArgueSampleTrace.
  Context `{params : CryptoParams}.
  Definition sk := Unop box_new (Const 0). (* secret key *)
  Definition pk := Unop box_keygen sk. (* public key *)
  Definition m := Symbolic EmptyString (* emptystring forces the message type *).
  Definition ct := Terop (@box _ string) sk pk m. (* ciphertext *)
  Definition ctx := Binop pair pk ct. (* the adversary's knowledge *)
  Example m_ind_ctxt : IND ctx m ctx.
    apply INDArgs; repeat split.
    - (* goal: IND ctx m pk *)
      repeat (apply INDArgs; repeat split).
    - (* goal: IND ctx m ct *)
      apply INDBoxMessage.
      (* goal: IND ctx sk ctx *)
      repeat (apply INDArgs; repeat split).
      (* goal: IND ctx sk m *)
      eapply INDBoxNewSym.
  Qed.
End ArgueSampleTrace.
```

3 Modeling Protocols

The IND relation provides a way of stating secrecy properties communications transcripts specified in its input language, but it does not say anything about what could have gone differently under different network delays or active attacks by an adversary. Similarly to looking at an arrows-and-messages diagram, proving that some transcript did not reveal secret information is not sufficient to conclude that the protocol always manages to keep the secrets secret. However, given a sufficiently powerful verification system such as Coq, the safety of the protocol can be verified by quantifying over all possible transcripts and proving the same IND statement. Fortunately, the Verdi framework for verifying properties of distributed systems includes a reusable core library for modeling concurrent interaction in a message passing system.

3.1 Concurrency and Authentication

To get a sense of how concurrency affects cryptographic reasoning, we an authentication property of a simple service using a model derived from Verdi. Verdi models a set of processes that receive input (from the outside world, "the user") and network messages (from other processes), and react to them by producing output and network messages. The framework also presents several models of the network, from stricter to more realistic: with or without message reordering, with or without message loss, with or without message duplication.

We added an additional rule to the network model that allowed for messages to be forged by an adversary. To retain the ability to prove any properties involving the network, we also added a third event type (in addition to input/output and message): a process either signs a message or verifies a signature by some other process. Using this, we stated the axiom that if a signature verification succeeds, a corresponding sign event must have occurred at some point in the past. While it would have been possible (and probably practical) to do so, in the interest of time, we did not update the Verdi programming language (a wrapper to two list monads) to include the crypto history but instead wrote our test program in S-expression form. Similarly, the current modification arbitrarily assumes that a process ignores all messages from which it finds invalid signatures.

We used this network model to verify the correctness of a small "time"-stamping server: for any message received from the network, the server would sign the message prepended with a value of its own. A client of this service would vet its reply by verifying the server's signature, and output (to the user) the content if the signature is correct. We proved that every output of the client indeed has the server's header prepended to it. The proof consisted of two lemmas, both by induction over "time" (the steps taken by the system): first, everything the server has ever signed has the correct header, and second, the client only outputs messages that were previously signed by the server. In our opinion, this reasoning is much more intuitive than an explicit reduction that an adversary who could make the client output something else must also be able to forge a signature. While the proof does use inversion to constraint where the contents of message must have originated, this bit of retroactive reasoning does not leak through to the statements of the lemmas. This enables modularity: extending the behavior of the client or the server would only require modifications to their corresponding lemma, not both. The following code snippet shows the modified rule in the network semantics and the statements of the two lemmas.

```
| SM_deliver : forall net net' p out d l cEvents,
   net_handlers (pDst p) (pSrc p) (pBody p) (nwState net (pDst  p)) =
                                            (out, d, l, cEvents) ->
```

```
     net' = mkNetwork (update (nwState net) (pDst p) d)
             ((mark_cevents_src (pDst p) cEvents) ++ (nwCrypto net)) ->
     (forall them msg, (In (Verify them msg) cEvents) ->
                    (In (them, Sign msg) (nwCrypto net'))) ->
     step_m net net' [(pDst p, inr out)]

Lemma signed_hasHeader : forall st trace,
    refl_trans_n1_trace step_m step_m_init st trace ->
    forall n s, In (n, Sign s) (nwCrypto st) -> hasHeader s.

Lemma notary_correct:
    forall st trace, step_m_star step_m_init st trace ->
    forall (n:Name) (ss:list string),
    In (n, inr ss) trace ->  forall s, In s ss -> hashHeader s
```

From the success of this experiment we concluded that the Verdi model of message passing can be modified to perform the concurrency-related reasoning needed for modeling cryptographic protocols. The Verdi proofs no longer hold because the network semantics need to be modified to allow for messages to be tampered with, but the general approach seems to be a very good fit.

3.2 Concurrency and Secrecy

To be able to argue secrecy of the network state tracked by Verdi core, we change its type from `list msg` to `Exp (list msg)` and modify the state transition rules to take an expression for the input state to an expression for the outputs and a new state. This requires modifying the expression language to support defining expressions that act on some abstract inputs and later substituting concrete values for the placeholder. We use the Parametric Higher Order Abstract Syntax technique[12] to implement substitution: instead of giving the placeholder a name, we *internally* model an underspecified expression as a function from an input *of any type* to an expression where each node is annotated with what it would be after substitution. All inputs that make the annotation of the top-level expression have the same type as the original expression must are guaranteed to be valid substitutions by the typing rules – code that must work for all types of the value being substituted cannot modify the value because it does not know its type. The current interface for specifying cryptographic protocols is as follows:

```
Class NetParams (P : CryptoParams) :=
  {
    input : Type ;
    output : Type ;
    state : Type ;
```

[12]http://adam.chlipala.net/cpdt/html/ProgLang.html

```
    msg : Type ; (* network message *)

    name : Type ;
    name_eq_dec : forall x y : name, {x = y} + {x <> y} ;
    nodes : list name ;
    all_names_nodes : forall n, In n nodes ;
    no_dup_nodes : NoDup nodes ;

    init_handlers : name -> Exp state;
    (* Exp1 T1 T2 is an expression of type T2 with one free variable of type T1 *)
    net_handlers : name -> Exp1 (state * msg) (state * list output * list msg);
    input_handlers : name -> Exp1 (state * input) (state * list msg)
  }.
```

As discussed earlier, it is necessary to modify Verdi's network semantics to model the increased uncertainty caused by the presence of an attacker: a message received by a good node is no longer required to have been sent by a good node. The limits on the computational power and luck available to the adversary are encoded in the restriction that the messages good nodes receive from the network must be inferable from what has already been revealed. For example, an attacker is not allowed to guess the secret key of a server and add it to the network context by feeding it into an echo service provided by that server. Constants, such as those that appear in the specification of the protocol, are however by definition indistinguishable from symbolic messages (that come from the user) and can be freely manipulated by the attacker. Furthermore, if an attacker has independently learned a message and a secret key, it is no longer true that an authenticated encryption of the message under that secret key is indistinguishable from other authenticated encryptions, so the adversary can use it to their advantage. Note that none of this is specified as an exhaustive list of what an attacker can do as in classical Dolev-Yao models; the only directly specified constraint is that the attacker cannot send out information that it does not know. The following code snippet shows the inductive definition of how the state of the network and the participants changes throughout the execution of a protocol and a statement of an example lemma.

```
  Record network := mkNetwork { nwCtx : Exp (list msg);
                                nwState   : name -> Exp state }.

    Inductive step_c : step_relation network (name * (input + Exp (list output))) :=
  | SC_deliver : forall net n m r,
         (~ IND m (nwCtx net) (nwCtx net)) ->
         Subst (Binop pair (nwState net n) m) (net_handlers n) = r ->
         step_c
```

```
          net
          (mkNetwork
            (Binop (@app msg) (Unop snd r) (nwCtx net))
            (update (nwState net) n (Unop fst (Unop fst r))))
                  [(n, inr (Unop snd (Unop fst r)))]
  | SC_input : forall net n i r,
        Subst (Binop pair (nwState net n) (Symbolic i)) (input_handlers n) = r ->
        step_c
          net
          (mkNetwork
            (Binop (@app msg) (Unop snd r) (nwCtx net))
            (update (nwState net) n (Unop fst r)))
          [(n, inl i)]
  .

Example longtermKeySecret : forall w trace, step_c_star step_c_init w trace ->
  IND (nwCtx w) (Unop state_longterm_sk (nwState w Alice)) (nwCtx w).
```

4 Limitations and Future Work

While the current expression language is technically general enough for encoding real-world protocols, it is not convenient enough to make doing so practical. The most important missing feature is lack of support for structured data: records are treated the same way as any other type, so the result of looking at one field in a record is tainted with the information from all of them. We tried to code around this while implementing SIGMA, but the resulting code was practically unreadable, let alone verifiable. Simply admitting the specific proof goals relating to independence of two fields of the same record would be both deeply dissatisfying and block any significant use of proof automation. Support for a structured data abstraction is as crucial for verifying protocols as it is for implementing them, and seems inevitably necessary for verification of realistic protocols. Additionally, support for a recursion or looping construct would be necessary for protocols that use iteration, for example Axolotl.

A more mundane change that would also improve usability would be to port the Verdi embedded language (Coq notation around list monads) to the expression language described in here. The `LockServer` example in the Verdi paper is simple when read in nice syntax yet barely recognizable in S-expression format, a similar change would have improved the experience of implementing SIGMA a lot.

The model of cryptographic primitives could be made more comprehensive and accurate by tracking a number of additional factors. Here is a probably incomplete list of the parts of NaCl that are not modeled in the current version of this project.

1. Information flow through length of ciphertext (would require restricting the plaintexts to types that have a defined length).
2. Information flow from the public keys to the ciphertext (non-anonymity).
3. Authentication properties of `crypto_box`.
4. Decrypt/Verify functions that do not abort (would require making the `IND` rules depend on the result of the operation as for `sel`).
5. Symmetric encryption and authentication (would probably be simpler than the public key counterparts and follow the same style).
6. Nonce management: the good properties of `crypto_box` hold only if the context includes only one distinct encrypted message per each nonce and unordered pair of the sender's and receiver's public keys.
7. "Scalar multiplication" for generating general purpose shared secrets (`IND sk1 e -> IND sk2 e -> IND (smul sk1 (smul sk2 base)) e`).

It would also be interesting to see if the hand-written specifications of the cryptographic primitives could be proven sound in a computational model, perhaps using CertiCrypt[13] or EasyCrypt[14]. This additional rigor may seem unnecessary, but formalizing cryptographic semantics is subtle: for example, modeling textbook RSA without taking into account that its ciphertexts are homomoorphic with respect to multiplication can let exploitable weaknesses to stay unnoticed.

5 Reflections on Engineering Verified Systems

1. One cannot verify what one does not understand.
2. Tooling quality matters.

Seriously, though, that's basically all that I have to say. In some more words…

The hardest issues I have tackled while practicing formal verification are those of my own ignorance: if I don't know how to model or prove a thing, there is no use having all world's rigor and fluency embodied in a piece of software. These kind of issues are the hard to recognize and even harder to admit because there always they usually show up when integrating pieces of code already deemed correct, and there is rarely a single line that somebody can point a finger at and hiss "bug!". When I verified Dijkstra's algorithm I made unidiomatic implementation choices because I did not understand how structural recursion and induction hypotheses relate to for loops and invariants, and when I finally got to cleaning up the mess I managed to write down a too specific invariant in place of a general one in the hurry. Both errors were conceptual, and started to show far away from their root cause. I think I did a better job of being aware of my limitations

[13]http://certicrypt.gforge.inria.fr/
[14]https://www.easycrypt.info/trac/

in this project, but the issues were still the same. Because I do not understand how dependent pattern matching works internally, I had no way of predicting that an embedded functional language calling functions by `hlist`-s of arguments would make it hard to reason about which element of a known list of arguments an unknown value is. Furthermore, the solution was completely manageable once known: isolate the smallest proof context where the issue shows, `Import Program` and use `dependent destruction` instead of `destruction`, and things work (thanks to Peng Wang for helping me out). Yet I did not know what I did not know, and there went those two weeks. Similarly, the current blocker (taint tracking over structured data) could have been foreseen if I had worked on or seriously thought about implementing any programming language with structured data, not just `Imp` and `lisp`. The fact that I did not encounter any serious crypto-related issues this time I would attribute to… well, having encountered them during my the previous attempts of building a similar project.

Even though I had a generally good time with the Coq system, I did encounter a handful of Anomalies. Most (maybe all) of them were results of me typing in blatant absurdities that I had managed to confound myself were correct, but it would have been so much nicer to receive an actual error message instead of "Anomaly" or "Universe inconsistency", or a crash. In particular, I found the `Inductive` definition system to be noticeably flaky in my April 2015 build of Coq 8.5 (the `Proof` mode was generally okay). Now, from a technical point of view, these problems *are* secondary when compared to conceptual errors, but they also have detrimental effect on learning and motivation. It is infinitely more satisfying to see an explainable error than a segfault, and being able to use a standard library model of a simpler expression language instead of making my own when would have been a good way of getting started, even though I might have needed to change it to fit my needs in the long run. There are enough papers that prove that there exists a way in which Coq can be used to verify real systems, but making it scale would require well-documented curated libraries of reusable building blocks. And there is a long way between "was good enough to get a result for a paper" and "fit to be in a standard toolchain". While our lack of understanding of our systems is the main bottleneck to getting them verified, the main obstacle to exposing our misconceptions is how frustrating it is to develop in Coq, which is mostly caused by lacking software engineering and technical debt. Well, at least these were my two cents.