Probabilistic Relational Verification for Cryptographic Implementations

Gilles Barthe1 Cédric Fournet2 Benjamin Grégoire3
Pierre-Yves Strub1 Nikhil Swamy2 Santiago Zanella-Béguelin2

IMDEA Software Institute1 Microsoft Research2 INRIA3
{gilles.barthe, pierreyves.strub}@imdea.org, {fournet, nswamy, santiago}@microsoft.com, benjamin.gregoire@sophia.inria.fr

Abstract

Relational program logics have been used for mechanizing formal proofs of various cryptographic constructions. With an eye towards scaling these successes towards end-to-end security proofs for implementations of distributed systems, we present rF*, a relational extension of F*, a general-purpose higher-order stateful programming language with a verification system based on refinement types. The distinguishing feature of rF* is a relational Hoare logic for a higher-order, stateful, probabilistic language. Through careful language design, we adapt the F* typechecker to generate both classic and relational verification conditions, and to automatically discharge their proofs using an SMT solver. Thus, we are able to benefit from the existing features of F*, including its abstraction facilities for modular reasoning about program fragments. We evaluate rF* experimentally by programming a series of cryptographic constructions and protocols, and by verifying their security properties, ranging from information flow to unlinkability, integrity, and privacy. Moreover, we validate the design of rF* by formalizing in Coq a core probabilistic λ-calculus and a relational refinement type system and proving the soundness of the latter against a denotational semantics of the probabilistic λ-calculus.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics; D.2.4 [Software Engineering]: Software/Program Verification.

Keywords program logics; probabilistic programming

1. Introduction

Many fundamental notions of security go beyond what is expressible as a property of a single execution of a program. For example, non-interference [25], the property underlying information-flow security, relates the observable behaviors of two program executions. Recognizing the importance to computer security of such hyperproperties [20], researchers have developed a range of program analyses and verification tools for proving relations between two or more programs, or two or more executions of the same program. For instance, Benton’s relational Hoare logic [11] generalizes Hoare logic to reason about properties of two programs.

In addition, security properties must often account for probabilistic behaviors. For instance, in cryptography, simulation- and indistinguishability-based notions of security are specified in terms of the probability that an adversary wins in some probabilistic experiment. Starting from Kozen’s seminal work [30], many logics for reasoning about probabilistic programs have been developed.

Recently, these two lines of work have been combined into a relational program logic called pRHL for reasoning about probabilistic imperative programs [7]. This logic can justify common patterns of probabilistic reasoning about hyperproperties used in cryptographic proofs, including observational equivalence, equivalence up to failure and reductionist arguments. pRHL forms the backbone of EasyCrypt [8], a tool-assisted framework which has been used for verifying the security of encryption and signature schemes, modes of operation for block-ciphers, and hash function designs in the computational model. These advances, among others, raise the prospect of a new class of verifiably secure systems: those that are proven secure based on standard computational assumptions (such as the existence of one-way functions) and whose verification encompasses all aspects of the system implementation.

rF*: end-to-end security of cryptographic implementations In an effort to scale logics like pRHL towards end-to-end security proofs of system implementations, this article presents a new language called rF*, building on F* [45], a dependently typed dialect of ML. F*, and its predecessor F7 [12], make use of refinement types to verify implementations scaling to tens of thousands of lines of code, including the Transport Layer Security Internet standard (TLS) [13], multi-party sessions, web-browser extensions, zero-knowledge protocols, and the F* typechecker itself.

rF* integrates within F* an expressive system of relational refinements to support fine-grained reasoning about probabilistic computations. Through careful language design, we are able to use the relational features of rF* in smooth conjunction with the existing features of F*, allowing the large corpus of already-verified F*-code to be reasoned about effectively when used in a relational context. As such, our work opens the door to semi-automated security verification through relational refinement types, instead of a somewhat inflexible combination of parametricity and type safety (as in F7), or through detailed tactic-based interactive proofs (as in EasyCrypt), as a basis for certifying the security of critical pieces of Internet infrastructure, such as a reference implementation of TLS.

Technically, this paper makes three broad contributions:
1. A relational logic for higher-order stateful probabilistic programs: we formalize in the Coq proof assistant $\lambda_p$, a lambda calculus with references, random sampling, and unbounded recursion. We develop a relational refinement type system for $\lambda_p$, and prove it sound with respect to a denotational interpretation of judgments as relations over pairs of store-passing probabilistic functions. To our best knowledge, $\lambda_p$ is the first relational logic for higher-order stateful probabilistic programs. (§3)

2. The design and implementation of $\mathsf{RF}^*$: $\lambda_p$ forms the basis of the design of $\mathsf{RF}^*$, an extension of $\mathsf{F}^*$. We show how to encode relational refinement types within a new relational state monad, $\mathsf{RDST}$. We provide a type inference algorithm for $\mathsf{RDST}$, in the form of a weakest pre-condition calculus that computes relational verification conditions. Proofs of these verification conditions can be discharged automatically by the $\mathsf{RF}^*$ typechecker and the $\mathsf{Z3}$ [21] SMT solver. (§4)

3. An experimental evaluation of $\mathsf{RF}^*$: we demonstrate the expressiveness of $\mathsf{RF}^*$ through a representative set of examples, starting from simple (non-probabilistic) information flow, and gradually moving towards more advanced cryptographic models and systems. To date, we have used $\mathsf{RF}^*$ to automatically verify a total of around 1,400 lines of code for a variety of relational properties, ranging from termination-insensitive non-interference to various indistinguishability-based properties for encryption, besides others. Several examples make essential use of higher-order and stateful features of $\mathsf{RF}^*$, emphasizing the utility of the $\lambda_p$ logic for practical security verification. (§2 and §5)

The $\lambda_p$ theory formalized in Coq, the $\mathsf{RF}^*$ compiler, and all the example programs mentioned in this paper are available online from http://research.microsoft.com/fstar.

2. Programming with relational refinements

We start by describing $\mathsf{RF}^*$ informally through a series of examples, beginning with a brief introduction to $\mathsf{F}^*$ itself, and then focusing on the main new feature in $\mathsf{RF}^*$, i.e., relational refinement types.

2.1 From classic to relational refinements. $\mathsf{F}^*$ is a call-by-value higher-order programming language with primitive state and exceptions, similar to ML, but with a more expressive type system based on dependent refinement types. Refinement types are written $\mathit{xt}(\phi)$ where $\phi$ is a logical formula. For instance, the code fragment below defines a refined type for non-negative integers, then for integers modulo some number $p$:

- $\mathsf{type} \ \mathsf{nat} = \mathsf{nint}\{0 \leq n\}$
- $\mathsf{type} \ \mathsf{mod} \ p = \mathsf{nint}\{n < p\}$
- $\mathsf{let} \ \mathsf{p} = 97$
- $\mathsf{let} \ n : \mathsf{mod} \ p = 73$

Typechecking $\mathsf{F}^*$ programs involves logical proof obligations, which are delegated to the $\mathsf{Z3}$ SMT solver. For instance, to check that $\mathsf{n}$ has type $\mathsf{mod} \ p$, the $\mathsf{F}^*$ typechecker emits the proof obligation $\mathsf{p} = 97 \implies 0 \leq 73 < p$, which is easily discharged by $\mathsf{Z3}$. Type safety means that, whenever an expression $e$ with type $\mathit{xt}(\phi)$ reduces to a value $v$, this value $v$ satisfies the formula $\phi(v) : \mathit{t}$.

The system provides structural subtyping. For instance, $\mathsf{nat}$ is a subtype of $\mathsf{int}$, and $\mathsf{mod} \ p$ is a subtype of $\mathsf{mod} \ q$ when $p \leq q$. These subtyping relations are automatically proved and applied by $\mathsf{F}^*$. Refinements can be combined with dependent function types, written $\mathit{xt} \rightarrow \mathit{t'}$, where the formal parameter $\mathit{xt}$ is in scope in the result type $\mathit{t'}$. We also use dependent pairs, written $\mathit{xt} \times \mathit{t'}$, where the variable $x$ of the first component is in scope in the type $\mathit{t'}$ of the second component. For instance, we may write and typecheck addition modulo as follows (where the refinement braces bind tighter than the arrow):

$$\begin{align*}
\mathsf{val} \ \mathsf{add} : & \mathsf{p:nat} \rightarrow \mathsf{x:mod} \ p \rightarrow \mathsf{y:mod} \ p \rightarrow \mathsf{z:mod} \ p \ \{z = (x + y) \mod p\} \\
\mathsf{let} \ \mathsf{add} \ \mathsf{pxy} = & \mathsf{let} \ \mathsf{s} = \mathsf{x + y} \ \mathsf{in} \mathsf{if} \ \mathsf{s} < \mathsf{p} \ \mathsf{then} \mathsf{s} \mathsf{else} \mathsf{s} \mathsf{- p} 
\end{align*}$$

$\mathsf{F}^*$ also provides primitive support for programming with state. For example, one may write $\mathsf{let} \ \mathsf{incr} = \mathsf{i} := \mathsf{i} + 1$. By combining refinements with references, one can express invariants on the program state, e.g., $\mathsf{ref nat}$ is the type of mutable locations that contain non-negative integers. To describe more precise properties of effectful programs, $\mathsf{F}^*$ provides more advanced mechanisms, including a monadic mode [46], where one can reason about programs using variants of the Hoare state monad of Nanevski et al. [34] together with McCarthy’s select/update theory for modeling the heap [32]. For example, one can give $\mathsf{inc}$ a specification of the form $\mathsf{ref nat} = \mathsf{ST}(\mathsf{Ax}:\mathsf{True})\ \mathsf{unit}(\mathsf{λ}\mathsf{h}:\mathsf{h'}:={\mathsf{Upd}}\mathsf{h}\mathsf{i}(\mathsf{+}\mathsf{Sel}\mathsf{h}\mathsf{i}))$, where $\mathsf{ST} \ \mathsf{pre} \ \mathsf{post}$ can be understood as the state-passing function type $\mathsf{h:heap}(\mathsf{pre} \mathsf{h}) \rightarrow \mathsf{xt} \times \mathsf{h'}:\mathsf{heap}(\mathsf{post} \mathsf{h} \mathsf{h'}))$ although, in reality, $\mathsf{F}^*$ provides primitive support for state. That is, the type of $\mathsf{inc}$ states a trivial pre-condition on the input heap $\mathsf{h}$, and a post-condition indicating that the final heap $\mathsf{h'}$ differs from $\mathsf{h}$ at the location $\mathsf{i}$, which is incremented. $\mathsf{F}^*$ provides type inference in the form of a higher-order weakest pre-condition calculus to help ease the burden of writing such precise specifications [46].

$\mathsf{RF}^*$ extends $\mathsf{F}^*$ with relational refinements: a type can (also) be decorated with a relational formula, placed within double braces $\{\ldots\}$, that specifies a joint property on pairs of values. Relational formulas can independently refer to the left and right values of every program variable in scope, using the projections $\mathsf{L}$ and $\mathsf{R}$, respectively; projections extend naturally to arbitrary formulas. Intuitively, for deterministic programs, type safety means that, whenever we obtain two results $\mathsf{e}_L$ and $\mathsf{e}_R$ by evaluating an expression $e : \mathit{xt}(\phi)$ in two contexts that provide well-typed substitutions for $e$'s free variables, then the formula $\phi(\mathsf{L} \mathsf{e} \mathsf{L} \mathsf{R})$ is valid. More generally, instead of considering two executions of the same program $e$, $\mathsf{RF}^*$ allows proving relations between the results of two programs, i.e., we relate $\mathsf{e}_0$ and $\mathsf{e}_1$ at (relationally refined) type using $\mathsf{e}_0 \sim \mathsf{e}_1 : \mathsf{t}$. We write $\mathsf{e} : \mathsf{t}$ as a shorthand for $\mathsf{e} \sim \mathsf{e} : \mathsf{t}$.

We start with a few simple examples. Take the expression $e$ to be $z - z$; we can give the type $\mathsf{x:xt}([\mathsf{L} \mathsf{x} : \mathsf{R} \equiv \mathsf{x}])$, meaning that for any pair of substitutions $\sigma_L$ and $\sigma_R$, evaluating $\sigma_L e$ yields the same result as evaluating $\sigma_R e$. Similarly, we have $\mathsf{x+} \sim \mathsf{2:xt}([\mathsf{L} \mathsf{x} \equiv \mathsf{R} \mathsf{x} \Rightarrow \mathsf{L} \mathsf{z} \equiv \mathsf{R} \mathsf{z}])$, stating a simple equivalence between two integer expressions evaluated with the same value for $\mathsf{x}$.

Relational refinements can also be used to describe properties beyond equivalence. For example, we can express the type of monotonic integer functions as $\mathsf{x:xt}([\mathsf{L} \mathsf{x} \equiv \mathsf{R} \mathsf{x} \Rightarrow \mathsf{L} \mathsf{y} \equiv \mathsf{R} \mathsf{y}])$ and the type of $k$-sensitive integer functions, for some metric $\mathsf{d}$, as $\mathsf{x:xt}([\mathsf{dist(L y)} \leq k \times \mathsf{dist(L x)(R x)})]$. $\mathsf{RF}^*$ can automatically check (by subtyping) that a function such as $\mathsf{fun} \mathsf{x:} \rightarrow \mathsf{k} \mathsf{x}$ is both monotonic and $k$-sensitive for any $k \geq 0$.

Relational refinements are strictly more expressive than plain refinements: one can encode any plain refinement $\{\phi\}$ as the relational refinement $\{\mathsf{L} \phi \land \mathsf{R} \phi\}$ that independently specifies left and right properties. For instance, the type $\mathsf{nat}$ above is automatically desugared to $\mathsf{nint}(\{0 \leq n \land n \not\equiv \mathsf{R} n\})$. Pragmatically, this enables us to mix property refinements and relational refinements in our concrete syntax, and to import any refinement-typed $\mathsf{F}^*$ library in relational mode by applying the encoding. When authoring programs with specific relational properties in mind, one needs to use a special compiler directive (aka a pragma) to switch the verifier to relational mode. We contend that the resulting language, $\mathsf{RF}^*$, brings relational program verification out of the domain of tools applied to small fragments of pseudocode with interactive proofs, to a practical programming language suitable for small- to medium-scale systems implementations.
2.2 Information flow. Relational refinements can be systematically used to give a semantic characterization of termination-insensitive non-interference [41]. Whereas standard type-based information flow controls resort to security labels and ad hoc syntactic mechanisms to conservatively determine when the observable outputs of a program may depend on its secret inputs, RF* can directly verify the corresponding equivalences, as illustrated below.

Recall that non-interference means that public results do not depend on secrets. If an expression e with base type a that computes over some secret information can be given the type eq a = x:a[[L x = R x]], then its result can be safely published, since the execution of e reveals no information about the secrets.

Capturing the intuition from labelled information flow type systems with “high” and “low” confidentiality levels, we use the type eq a (the type of values that are “equal on both sides”) for low-confidence values, also written low a. In contrast, we use the type a (the type of unrelated left and right values) for high-confidence values, writing hi a as an alias for a. As usual, low a is a subtype of hi a, meaning that public values can be treated as secret, but not the converse.

Using these type abbreviations, we can write programs such as fun (x,y) → (x+y, y+1) and give them flow information types such as hi int × low int → hi int × low int. More interestingly, we can supplement these types with relational refinements that capture more flexible information-flow policies. For example, a plausible confidentiality policy for credit card numbers conceals all but their last four digits, as specified and implemented below.

```
val last4 : n:hi int → string =  |_{L n%10000 = R n%10000}⇒ L s = R s |
let last4 n = "********" ° int2string (n % 10000)
```

Tracking leaks via control dependencies (aka implicit flows) is a characteristic feature of information flow type systems. To illustrate how RF* reasons about implicit flows, consider the program fun b → if b then e else e′. Assuming that b is secret, flow-insensitive type systems would conservatively give this program the type hi bool → hi a. To give this program the more precise type hi bool → low a, we need to analyze four cases that arise from applying this function twice to arbitrary boolean arguments L b and R b, and prove that the results in all cases are the same. The four typechecking goals are

- e:low a, assuming L b = R b = true;
- e':low a, assuming L b = R b = false;
- e = e':low a, assuming L b = true and R b = false; and
- e' = e:low a, assuming L b = false and R b = true.

Anticipating on the next section, our proof rules for relating two values v0 and v1 are relatively simple. Proving v0 ∼ v1 : c×t[|φ|] involves first proving v0 ∼ v1 : t (which for base types involves simply showing that both v0 and v1 have type t), and then proving φ[L x, R x := v0, v1]. So, RF* can easily prove, for instance,

```
(fun b → if b then 0 else 0) : hi bool → low int
   and
   (fun x → if x=0 then x else 0) : hi int → low int
```

even though, syntactically, those functions branch on confidential values. For expressions, particularly those that have side effects, the problem is more complex. Our strategy is to adapt the Hoare monad ST pre t post provided by F* to a relational version called RST, where RST pre t post can be seen as the type shown below:

```
h:heap[| pre (L h) (R h) |] → c×t + h:heap[| post (L h) (R h) (L x) (R x) (L h') (R h') |]
```

This is the type of pairs of functions that, when run in a pair of input heaps L h and R h satisfying the 2-place relational pre-condition predicate pre, may diverge, but if they both converge, yield results L x and R x and output heaps L h' and R h' that satisfy the 6-place relational post-condition predicate post.

Using the RST monad and its associated weakest pre-condition calculus, we can type the following program, which branches on a confidential value and then performs matching public side-effects:

```
val f : x:ref int → b:bool → RST (λ x → unit) pre t post
   where post h0 h1 → h0' = let h0 = h0 × h1 in h0' = h0'
   let f x = b
   if b then x := 1 else x := 1
```

RF* infers a weakest pre-condition predicate transformer for this program, then checks that it is consistent with any programmer supplied annotation (the annotation is optional for loop-free programs). In our example, the pre-condition of f states that it can be run in any pair of heaps, while its post-condition ensures that, if b is applied twice to the same references in the same heaps, then regardless of its boolean argument, the resulting heaps are also the same, i.e., the type reveals that f does not leak information despite having side-effects guarded by a secret boolean.

More complex programs, for example those that may leak information via side-effects based on aliasing, can be verified similarly:

```
val g : x:ref int → y:ref int → b:bool → RST (λ x → unit) pre t post
   where post h0 h1 → h0' = L x=0 Y ∧ R x=0 Y → h0' = L y=0 Y ∧ R y=0 Y
   let g x y = b
   if b then x := 1 else y := 1; x := 0
```

The type of g states that, if x and y are not aliased, then the final contents of the reference y are the same.

Thus, the expressiveness of RF*, combined with its ability to use Z3 to discharge proof obligations, enables automated reasoning in the style of a relational Hoare logic for proving non-interference properties of higher-order stateful programs.

2.3 Sampling, chosen-plaintext security, and one-time pads. We introduce probabilistic relational reasoning in RF* using symmetric encryption schemes. Our goal is to communicate encrypted messages between a sender and a receiver without leaking any information about their content. (From an information flow viewpoint, ciphertexts are public, whereas plaintexts are secret.) For simplicity, we assume that messages range over byte arrays with a fixed size n (called blocks) and we do not consider active attackers. Padding and authentication against chosen-ciphertext attacks can be easily added, but would complicate our presentation (see §5 for a description of our model for chosen-ciphertext security).

We assume that the sender and the receiver share a secret key k (also a block), sampled uniformly at random by calling the primitive function sample. We model this assumption by writing a single program where both parties are within the scope of this key. The simplest secure encryption scheme is the one-time pad, implemented, for instance, using bitwise XOR: to encrypt the message p, compute c = k ⊕ p; to decrypt c, compute p = k ⊕ c.

```
type block = b:bytes { Length b = n }
let encrypt k p = xor k p
let decrypt k c = xor k c
```

Next, we explain how to specify and prove that an encryption scheme is secure. In cryptography, confidentiality is usually stated as resistance against chosen-plaintext attacks (CPA) and encoded as a decisional game in which an adversary chooses two plaintexts, receives the encryption of one of them under a fresh key, and must guess which of the two plaintexts was encrypted. (Decryption plays no role in this simple game; still, we may typecheck that it undoes encryption using classical refinements and properties of XOR.) This game may be coded in RF* as follows:

```
let CPA b p0 p1 = let p = if b then p0 else p1 in encrypt (sample n) p
```

where b is private and p0, p1, and the result are public. We thus express (perfect) CPA security relationally with the following type:
val cpa: b:bool → eq block → eq block → eq block

stating that the encryption of one of the two chosen-plaintext blocks
ρ0 or ρ1 depending on b does not leak any information about b,
hence does not help the adversary to win the game.

In fact, viewing CPA security from an information flow perspective,
a simpler formulation is possible. Instead of reasoning about
two messages selected by b, we just need to show that the function
\text{let } cpa' p = encrypt sample n p \text{ has the type } block → eq block.
This is the best type we can hope for encryption, treating the plaintext
as private and the ciphertext as public. This more compact typing
property subsumes the first one.

To prove secrecy for the one-time pad, some probabilistic rea-
soning is called for. Indeed, operationally, calling sample n twice
does not usually return the same value. However, relying on our
formal semantics, we show that it is permissible to give sample n a
more specific relational type that allows us to complete the proof.
In particular, as explained below, we can type the call to sample n
in a way that depends on the plaintext p and give it the type

\text{m:} \text{block } \{ \{ \text{xor} \ (L \ p) \ (L \ m) = \text{xor} \ (R \ p) \ (R \ m) \} \}

From this type, RF+ automatically proves cpa': block → eq block.
Intuitively, this relational refinement is sound inasmuch as the dis-
tribution of the resulting ciphertext is independent of the plaintext.

In programs that contain sample, the interpretation of assertions
in the RST monad becomes probabilistic. We formalize this in §3,
but provide some intuition for their meaning here. If we can give the
type RST (λ, True) ρ ρ0 ∼ ρ1, then our logic guarantees that if Q
is an equivalence relation partitioning \( \mathbb{N} \) in a set of equivalence
classes \( \mathcal{S} \), and if running \( e_p \) in a heap \( h_0 \) reduces to \( v_0 \) and \( v_1 \) in
\( h_1 \) reduces to \( v_1 \), then for any \( \{ V \in \mathcal{S} \mid v_0 \in V \} \) is equal to
the probability that \( v_0 \in \mathcal{S} \). Similar conclusions can be
drawn in general. For example, if \( Q = h_0 \ x_0 \ x_1 \ h_0' \ h_1' \) implies
\( \text{P0} \ x_0 \rightarrow \text{P1} \ x_1 \), then the validity of the above assertion implies
that \( \text{P1} \ x_1 \). From this interpretation, one should be able to see that the re-
lational refinement on the result of sample in RF+ is not specific to
XOR. More generally, sample can be given a specificiation to state
that any two calls to it return a pair of values related by any given
one-to-one function on its range. Intuitively, relational refinements
in RF+ capture relations between the \textit{distribution over values} gen-
ated by probabilistic programs, rather than between values obtained
in specific executions. The relational typing of sample is valid since
applying a one-to-one function to a value uniformly chosen from a
discrete set does not change its distribution.

To reflect these general properties of uniform sampling, our
library provides a polymorphic, typed variant of sample, that takes
as additional ghost parameter F, a binary predicate on sampled
values of type block, whose refinement states that F must be an
injective function (or, equivalently, a bijection). This parameter has
\text{kind } \text{block } \Rightarrow \text{E}, \text{ where } \text{E} \text{ is the kind of ghost refinements in}
F+ (types in E are erased at runtime). In the RF+ standard library,
sample is typed as follows

\text{type } \text{Function } \text{F} = \text{∀a: } \text{b:F a } \land \text{∀a b1 b2:F a b1 } \land \text{∀a b2 } \Rightarrow \text{b1=b2}
\text{type } \text{Injective F = \text{Function } \text{F} } \land \text{∀a } \text{a2 b:F a1 } \land \text{F a2 } \Rightarrow \text{a1=a2}
\text{val sample:VF.\text{len}nat(Injective F) } \rightarrow \text{\text{block len}([\text{F} (L \ b) (R \ b)])}

In our one-time pad example, when calling sample \( n \) in cpa’, we
stantiate F to \( \lambda b. b \. \text{xor} \ (L \ p) \ (L \ m) = \text{xor} \ (R \ p) \ (R \ m) \), which is indeed
injective. In §5, we describe security proofs of more realistic en-
cryption schemes based on variants of the above typing for sample.

2.4 Implicit flows and passport linkability. Before justifying
our typing rules, notably for sample, we present a concrete pro-
tocol for RFID-equipped passports, implemented in RF+, and
we discuss a linkability attack against this protocol recently uncovered

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Passport Tag} & (physically sharing \( k \)) & \textbf{Reader} \\
\hline
\text{sample nt} & \text{nt} & \text{sample nr, kr} \\
\hline
\text{decrypt} & \{nt ∥ nr ∥ kr\}_k & \text{verify MAC} \\
\text{verify MAC} & \text{nt} & \text{nt} \\
\text{sample kt} & \{nr ∥ nt ∥ kt\}_k & \text{decrypt} \\
\hline
\end{tabular}
\caption{Basic Access Control protocol for passports by Chothia and Smirnov \cite{19}. We refer to their work for a detailed discussion. This attack is representative of common weaknesses in cryptographic implementations due to implicit flows in the handling of errors while processing decrypted data.}
\end{figure}

Following the ICAO specification for machine-readable travel
documents, all recent European passports embed RFID tags featur-
ing the Basic Access Control protocol, outlined in Fig. 1. The pro-
tocol has two roles, a passport tag and a reader, exchanging mes-
sages using short-distance wireless communications. The goal of
the protocol is to establish a shared session key for accessing bio-
metric data on the passport. Each passport tag has a unique key \( k \);
the reader derives this key from information obtained by scanning
the passport. (In reality, there is a negligible chance of two pass-
ports having the same key, because the key is derived from a hash
of this information.)

The passport first samples a 64-bit nonce \( nt \) and sends it as a
challenge to the reader. The reader samples its own nonce \( nr \)
and some keying materials \( kr \), then encrypts the concatenation of these
three values using \( k \). Concretely, the protocol implements authen-
ticated encryption as triple-DES encryption concatenated with a
plaintext Message Authentication Code (MAC). The passport de-
crypts, recomputes and checks the MAC to ensure that the message
has not been tampered with, then compares the received nonce \( nt \)
with the challenge, to confirm that the reader responded correctly.
If both checks succeed, it generates its own keying materials \( kt \),
appends them to the concatenation of the two nonces (in a different
order than before), and computes the session key \( s = kt ∥ kr \).
The reader then similarly decrypts the received ciphertext, checks
the MAC, and computes \( s \).

The code the tag uses for handling the encrypted message of the
reader is shown below; encrypt and decrypt provide authenticated
encryption; concat and split convert between triples of 64-bit values
and their concatenation.

\text{let tag1 k nt c = match decrypt k c with}
\quad Some p → let (nt',nr,kr) = split p in
\quad if nt = nt' then encrypt k (concat nr nt (sample,kt())
\quad else nonceError
\quad | None → decryptError

The code either produces an encrypted message, or it returns an
error code. As written, it enables the following linkability attack:

1. The attacker eavesdrops any run of the protocol between a
target passport and an honest reader, and records their second
message.
2. Later, to test the local presence of this passport, the attacker
runs the protocol (as the reader), replays the recorded message,
and observes the response: although the protocol always fails to
establish a key, the tag returns a nonceError if the two passports
are the same, and a decryptError otherwise.

Experimentally, French passports reliably return different error
messages, whereas other European passports return the same er-
ror message, but with measurably different timings. Although our approach does not directly catch timing attacks, those attacks can be conservatively analyzed by treating error codes produced at different code locations as distinct.

We interpret the above attack as an implicit flow of information from the key used to decrypt to the error message. Indeed, if we type the key \(k\) as high confidentiality and the nonce \(n\) and the cipher \(c\) with \(eq\) refinements (since they are exchanged on a public network), relational typechecking fails on the body of \(\text{tag1}\). The result of the decryption is (a priori) not the same on both sides, so the cross-cases involve proving, for instance that when decryption returns \(\text{Some p}\) on the left and \(\text{None}\) on the right, the two resulting expressions are equal, which fails on the proof obligation \(\text{nonceError = decryptError}\).

By ensuring that the same error messages are returned in both cases (i.e., by requiring that \(\text{nonceError = decryptError}\)) this case is prevented, but this alone is not sufficient for verifying the code. Naively, the cross-cases that arise when verifying the nested conditionals require proving, under a suitable relational path condition, that the encryption on the third line is indistinguishable from the error messages—which is patently false. However, by reflecting several cryptographic assumptions into detailed typing invariants in the protocol implementation, we can prove that such problematic cross-cases never actually arise (i.e., the path conditions guarding these cases are infeasible) and we can verify that this code preserves unlinkability. Specifically, we assume that the encryption is CPA, key-hiding, and CTXT (all specified by typing) and that there are no nonce collisions (the probability of a collision is less than \(q^{2^{-64}}\) where \(q\) is the number of sessions observed by the adversary). Arapinis et al. [5] also analyze the corrected protocol, using the applied \(\pi\)-calculus, essentially proving unlinkability in a more abstract, symbolic model of cryptography. Please refer to our online materials for a full listing and further discussion of the specification and implementation of \(\text{tag1}\).

3. Formal development

We formalize a core of \(\mathcal{RF}^*\) in the Ceq proof assistant by developing \(\lambda_{\mathcal{P}}\), a minimal higher-order language with (statically allocated) references, probabilistic assignments, and unbounded recursion. The formalization is based on the SRESS-S reflection extension [26], and on the ALEA library for distributions [6]. Overall, the formalization comprises over 2,500 lines of code excluding the aforementioned libraries.

The formalization is built in two steps. First, we consider a simply typed system \(G \vdash_{\mathcal{P}} e : T\) for \(\lambda_{\mathcal{P}}\). Simple types \(T\) are extended to relational refinement types \(\mathcal{C}\) where one can add relational pre- and post-conditions to function types. This allows us to define a relational type system \(G \vdash_{\mathcal{C}} e_0 \sim e_1 : \mathcal{C}\) that relates a pair of expressions \(e_0\) and \(e_1\) in the type \(\mathcal{C}\) under the relational context \(G\).

We then give a denotational semantics for well-typed judgments. Simple types are given a coprogram interpretation \([T]\) in the standard way. Judgments \(G \vdash_{\mathcal{P}} e : T\) are interpreted as the elements of the form \([\Phi\,\gamma]\), where \(I\) is any valuation for the context \(G\). Taking into account that \(\lambda_{\mathcal{P}}\) is a language with references and probabilistic assignments, the denotation \([\Phi]\) of \(e\) is defined as a function from memories (equivalently, states or heaps) to distributions over pairs composed of a memory and an element of \([T]\); we denote by \(\mathcal{M}([T])\) this function space. Relational types \(\mathcal{C}\) are interpreted as a binary relation \([\mathcal{C}]\) over \(\mathcal{M}([T])\), where \(T\) is the simple type derived from \(\mathcal{C}\) by erasing all refinements. This allows us to interpret (Theorem 1) a valid judgment \(G \vdash_{\mathcal{C}} e_0 \sim e_1 : \mathcal{C}\) by all the pairs of the form \((\{\Phi\}_{I_L}, \{\Phi\}_{I_R})\) in \([\mathcal{C}]\) for any pair of valuations \((I_L, I_R)\) for the erasure \(G\) of \(\mathcal{C}\).

### Figure 2. Well-formed relational formulas (excerpt)

| \(x : T \in G\) & \(\tau \in \{L, R\}\) & \(x : T \in G'\) |
|---|---|---|
| \(\neg \sigma \vdash_{\mathcal{P}} x\) & \(\tau \in \{L, R\}\) & \(\neg \sigma \vdash_{\mathcal{P}} x\) |
| \(\neg \sigma \vdash_{\mathcal{P}} \text{ref } B\) & \(\tau \in \{L, R\}\) & \(\neg \sigma \vdash_{\mathcal{P}} \text{ref } B\) |

3.1 \(\lambda_{\mathcal{P}}\): syntax. \(\lambda_{\mathcal{P}}\) is a simply typed \(\lambda\)-calculus with references and probabilistic assignments. For simplicity, we only consider two forms of probabilistic assignments: assigning a uniformly sampled boolean to a boolean variable (flip), and assigning an integer value sampled uniformly in a non-empty interval \([i, j]\) to an integer variable (pick\(i\)). Formally, the sets of types, contexts, values and expressions are given by the following grammars:

<table>
<thead>
<tr>
<th>Type</th>
<th>Context</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>(\mathcal{B})</td>
<td>(\mathcal{B})</td>
</tr>
<tr>
<td>(\mathcal{ctx} G)</td>
<td>({x : T})</td>
<td>({x : T})</td>
</tr>
<tr>
<td>(\text{value } v, u)</td>
<td>(c \mid {x \mid {o(v_1, \ldots, v_n)} \mid \text{fun } x : T \to e} \mid v)</td>
<td>(c \mid {x \mid {o(v_1, \ldots, v_n)} \mid \text{fun } x : T \to e} \mid v)</td>
</tr>
<tr>
<td>(\text{expr } e)</td>
<td>(v\mid v \mid \text{let } x = e_1 \in e_2)</td>
<td>(v\mid v \mid \text{let } x = e_1 \in e_2)</td>
</tr>
</tbody>
</table>

where \(x\) ranges over a set var of variables, \(r\) ranges over a set ref of references and \(o\) ranges over a set \(\mathcal{O}\) of \(\mathcal{B}\)-sorted operators, whose signature is of the form \(\mathcal{B}_i \times \cdots \times \mathcal{B}_n \to \mathcal{B}_m\). We assume that \(\mathcal{B}\) contains the unit type \(\mathcal{U}\) (unit) along with the types of Booleans \(\mathcal{B}\) and integers \(\mathcal{I}\). Their associated constructors are \(\text{true}, \text{false}\) and \(n\) for \(n \in \mathcal{N}\). We implicitly assume that each reference has an ambient base type, and write \(r : \text{ref } B\) to denote that \(r\) is a reference with base type \(B\). The dynamic semantics is defined in the standard way as the compact closure (for a call-by-value convention) of the \(\beta\)-contraction.

3.2 \(\mathcal{P}\): typing. As usual, a typing context is a sequence of bindings \(x : T\) such that the bound variables are pairwise distinct. The typing rules for deriving valid judgments \(G \vdash_{\mathcal{P}} v : T\) and \(G \vdash_{\mathcal{P}} e : T\), in a simply typed setting, are standard and omitted.

Relational refinement types. Relational assertions are formulas over tagged variables \(x_L\) or \(x_R\) and tagged references \(r_C\) or \(r_R\); informally, tags determine whether the interpretation of \(x\) or \(r\) will be taken w.r.t. the left or right projection of a relational valuation. In order to interface with automated first-order provers, relational assertions are first-order \(\mathcal{B}\)-sorted formulas built from operators in \(\mathcal{O}\) and predicates taken from a set of \(\mathcal{B}\)-sorted predicates that includes at least the equality predicates for all the base types. Note that tagged variables always occur free in assertions, and only logical variables can be bound. For instance, the relational assertion \(\forall y : \mathcal{I} \mid x_L \leq y \Rightarrow x_R \leq y + r_R\) is well-formed under any context \(G\) such that \(x : \text{int } \in G\), assuming that \(r : \text{ref } \mathcal{I}\).

Formally, relational assertions are defined using a first-order type system with judgments of the form \(G \vdash_{\mathcal{P}} \Phi\) for formulas (resp. \(G \vdash_{\mathcal{P}} v : T\) for values) where \(G\) (resp. \(G'\)) is a context for relational variables (resp. for non-relational variables introduced by quantifiers). Figure 2 shows the typing rules for variables, references, equality, and universal quantification. We say that an assertion \(\Phi\) is well-formed in context \(G\) iff \(G \vdash_{\mathcal{P}} \Phi\).

Refinement types are either relational types (denoted by \(\mathcal{T}, \mathcal{U}, \mathcal{V}\)), which will be used for relational typing of values, or computation types (denoted by \(\mathcal{C}\)), used for relational typing of expressions.
They are defined by the following grammar:

\[ \mathcal{T}, \mathcal{U}, \mathcal{V} := \mathbf{B} \mid (x : T) \rightarrow \mathcal{C} \quad \mathcal{C} := \{ \Phi \} x : \mathcal{T} \{ \Psi \} \]

where \( \Phi \) and \( \Psi \) are relational assertions. By convention, \( x \) is bound in \( (x : T) \rightarrow \mathcal{C} \) and \( \{ \Phi \} x : \mathcal{T} \{ \Psi \} \), and can be free in \( \mathcal{C} \) or \( \Psi \).

However, the type system enforces that \( x \) occurs in \( \mathcal{C} \) and \( \Psi \) only if \( T \) is a base type. In other cases, we write \( T \rightarrow \mathcal{C} \) and \( \{ \Phi \} x : \mathcal{T} \{ \Psi \} \).

A relational context \( G \) is a sequence of bindings \( x : T \) such that the post-bound variables are pairwise distinct.

The refinement type \( T \) is a refinement of \( T \) under \( G \), written \( G \vdash T \triangleleft T \), if \( T \) is the result of erasing all pre- and-post-conditions occurring in \( T \) and if any assertion \( \Phi \) that appears in \( T \) is well-formed in \( G \) augmented by the local context of \( \Phi \) in \( T \). This relation is extended to relational contexts: \( G \triangleleft A \) is the smallest relation such that \( \{ \} \triangleleft \) and if \( \Phi \vdash T \) and \( G \vdash T \triangleleft T \) then \( G, x : T \vdash A, x : T \).

### 3.3 \( \lambda_\phi \): denotational semantics.

**Background.** The denotational semantics of well-typed expressions is based on the sub-probability monad over sets [6, 37] and its generalization to complete partial orders (cpos); recall that a cpo is a partial order in which every ascending chain has a supremum. The unit interval \([0, 1]\) has a cpo structure w.r.t. the natural order on reals; moreover, every set can be lifted to a flat cpo by adding a bottom element, and the set of functions \( A \rightarrow B \) between a set \( A \) and a cpo \( B \) can be given the structure of a cpo. A function between two cpos is monotonic if it is order-preserving, and it is continuous if it is monotonic and preserves suprema. Given two cpos \( A \) and \( B \), we let \( A \triangleright B \) and \( A \rhd B \) denote the set of monotonic and continuous functions from \( A \) to \( B \) respectively.

By Kleene’s fixed point theorem, every continuous function \( f \) on a cpo has a least fixed point \( f \). A discrete sub-distribution over a set \( X \) is a continuous functional \( \mu : (X \rhd [0, 1]) \rightarrow [0, 1] \) that satisfies properties of linearity and convexity with inverse, and discreteness. In particular, the latter axiom states that the support \( \text{supp}(\mu) \) of \( \mu \), consisting of all elements \( x \in X \) such that \( \delta_x > 0 \), is discrete—where \( \delta_x \) denotes the Dirac function for \( x \) i.e., \( \delta_x(x) = 1 \) and \( \delta_x(y) = 0 \) if \( x \neq y \).

We let \( D(X) \) be the set of discrete sub-distributions over \( X \). \( D(X) \) has the structure of an \( \omega \)-complete partial order. Moreover, sub-distributions can be given the structure of a monad; the unit and composition operators are denoted by \( \eta \) and \( \eta \) respectively.

The relational interpretation of types rests on an operator \( \text{lift} \) that lifts relations over \( A \times B \) into relations over \( D(A) \times D(B) \); the operator is inspired from early works on probabilistic bisimulations [28] and is used in CertiCrypt [7] and EasyCrypt [9] to interpret relational judgments. Formally, let \( \mu_1 \in D(A) \) and \( \mu_2 \in D(B) \); then \( R^2 \mu_1 \mu_2 \) iff:

\[ \exists \mu : D(A \times B). \exists \mu_1, \mu_2(\mu) = \mu_1 \land \mu_2 \land \text{supp}(\mu) \subseteq R \]

where \( \pi_1 \) and \( \pi_2 \) are the projections for distributions over pairs, i.e.,

\[ \pi_1(\mu) = \text{bind } (\lambda (x,y). \text{unit } x) \]
\[ \pi_2(\mu) = \text{bind } (\lambda (x,y). \text{unit } y) \]

A fundamental property of this lifting operator is that given \( f : A \rhd [0, 1] \) and \( g : B \rhd [0, 1] \) such that \( \forall a \in A, b \in B. R \ a \ b \implies f \ a = g \ b \)

then \( R^2 \mu_1 \mu_2 \) implies that \( \mu_1 \ f = \mu_2 \ g \), i.e., the expected values of \( f \) in \( \mu_1 \) and \( g \) in \( \mu_2 \) coincide. Note that events can be viewed as \([0, 1]\)-valued functions, and in this case expectation coincides with probability.

**Interpretation.** We first provide a non-relational interpretation of valid judgments. We assume each base type \( \mathbf{B} \) is interpreted as a (flat) cpo \([B]\), and that each constructor \( c \) belonging to the base type \( \mathbf{B} \) is given a denotation \( B(c) \in [B] \). We define the set of semantical values as \( \bigcup_B [B] \), and then the set \( M \) of states as the set of well-typed mappings from references to semantical values.

\[ M = \{ m : \text{ref} \rightarrow \bigcup_B [B] \mid \forall r : \text{ref} B. m(r) \in [B] \} \]

Then we extend the interpretation to functional types by setting

\[ [T_1 \rightarrow T_2] = [T_1] \rhd M([T_2]) \]

where \( M(X) \rhd M \rightarrow D(X \times M) \).

A valuation \( I \) is a function that maps every declaration \( x : T \) to a semantical value. A valuation \( I \) is well-formed for \( G \), written \( I \vdash G \), if \( I \) maps every declaration \( x : T \) in \( G \) to an element of \( T \).
The following theorem states that all judgments of the logic are sound w.r.t. their interpretation; it implies that typing can be used to verify probabilistic claims, thanks to the properties of lifting.

**Theorem 1 (Soundness)**

- If $\mathcal{G} \vdash v_1 \sim v_2 : T$, then $\mathcal{G} \vdash v_1 \sim v_2 : T$.
- If $\mathcal{G} \vdash e_1 \sim e_2 : C$, then $\mathcal{G} \vdash e_1 \sim e_2 : C$.
\[
\begin{align*}
\{ e \}_{I} & = B(c) \\
\{ x \}_{I} & = I(x) \\
\{ \text{fun } x : T \rightarrow e \}_{I} & = \lambda d. \lambda m. \{ e \}_{I}'[x:=d] \\
\{ \text{lbind } v : T \rightarrow e \}_{I} & = \text{unit } (\{ v \}_{I}, m) \\
\{ \text{rbind } v : T \rightarrow e \}_{I} & = \text{bind } (\{ e \}_{I})' (\lambda f. \lambda m. f (\{ v \}_{I}, m)) \\
\{ \text{let } x = e_1 \text{ in } e_2 \}_{I} & = \text{bind } (\{ e\}_{I})' (\lambda d. \lambda m. \{ e_2 \}_{I}'[x:=d]) \\
\{ \text{letrec } f = e_1 \text{ in } e_2 \}_{I} & = \text{bind } (\text{fix } F) (\lambda d. \lambda m. \{ e_2 \}_{I}'[f:=d]) \\
\{ r := e \}_{I} & = \text{unit } (m(r), m) \\
\{ \text{if } e \text{ then } p \text{ else } q \}_{I} & = \text{bind } (\{ e \}_{I})' (\lambda d. \lambda m. (\text{fst } m [r := d])) \\
\{ \text{High} \}_{I} & = \text{bind } \text{Us } (\lambda b. (b, m)) \\
\{ \text{pick} \}_{I} & = \text{bind } \text{Us } (\lambda n. (n, m))
\end{align*}
\]

where \( F = \lambda d_1. \lambda d_2. \lambda m. (M)'_{I}[f:=d_1][g:=d_2]. \)

Figure 4. Interpretation of values and expressions

Technically, we prove the soundness of each rule as a lemma, directly from the semantics. It allows us to fall back to the full generality of Coq whenever reasoning outside of the logic is required.

4. Encoding \( \lambda_p \) in \( R F^* \)

We now discuss our language design and implementation. There are two key ideas behind our encoding of \( \lambda_p \) in \( R F^* \). First, as shown in §2.3, we introduce probabilistic computations into \( F^* \) axiomatically, by providing a sample primitive at the appropriate type. Programmers can instantiate sample at runtime by providing a suitable source of randomness. Next, as discussed in §2.2, we adapt the Hoare state monad \( S T \) to a monad \( R S T \) for computations with relational pre- and post-conditions. We provide here more details about our encodings, in particular the style we adopt to compute relational VCs for the \( R S T \) monad, and the manner in which we reuse classical specifications.

4.1 Representing \( \lambda_p \) types. To implement \( \lambda_p \), we begin with a translation of its types into \( F^* \) augmented with a relational state monad. To stay close to \( \lambda_p \), our translation uses a monad \( R S T_0 \), which we then adapt to the monad \( R S T \) of §2. Like in \( \lambda_p \), post-conditions in \( R S T_0 \) only relate the output values and heaps, not the initial heaps. Specifically, the type \( R S T_0 \) of a post can be interpreted in \( R F^* \) as a store-passing function (over a primitive heap) with the signature shown below:

\[
\text{RST}_0 \text{pre a post = h:heap}(\text{pre } (L \ h) (R \ h)) \rightarrow (x:a \mapsto h:heap)(\text{post } (L \ x) (R \ h'))(R \ h''))
\]

The type translation is homomorphic on most of \( \lambda_p \)'s typing constructs, with the interesting cases being the cases on the computation types, where \( \{ \Phi \}_{I}[T : \Psi] \) is \( \text{RST}_0 \{ \Phi \}_{I}[T : \Psi] \) (\( \lambda h : h', \Psi \)).

4.2 A monad of predicate transformers for VC generation.

Next, to provide type inference for \( R F^* \), rather than writing relational Hoare triples in \( R S T \), we write specifications using predicate transformers. This style is adapted from the \textit{Dijkstra state monad}, previously introduced for inferring classical (non-relational) verification conditions for stateful \( F^* \) programs [46]. In particular, we introduce the \textit{relational} Dijkstra state monad, \( R D S T \), and show its signature below. (We write polymorphic types implicitly assuming their free type variables are prefixed quantified.)

\[
\text{RDST } \text{a wp = } \text{RST}_0 \text{(wp p) a p}
\]

The type \( \text{RDST} \) \( t \) \( wp \) is an abbreviation for the \( \text{RST}_0 \) monad that is polymorphic in its post-condition. Specifically, \( \text{RDST} \) \( t \) \( wp \) is the type of computation which for any relational post-condition \( p \) on \( t \) and heaps, the pre-condition on the input heaps is given by \( wp \).

Unlike the Hoare-style \( R S T_0 \) monad, the \( R D S T \) monad yields a weakest pre-condition calculus by construction. As indicated by the signature of \( \text{bind} \), when composing computations in the \( R D S T \) monad, we simply compute a pre-condition for the computation by composing the predicate transformers of each component. A slight complication comes from the need to constrain the formal parameter \( x : a \) of \( wp \) \( 2 \) relationally. In general, \( wp \) \( 2 \) will have free occurrences of \( L \ x \) and \( R \ x \). We relate these to the result of the first computation using the guard \( L \ x = 0 \) and \( R \ x = 1 \), before composing \( wp \) \( 1 \) and \( wp \) \( 2 \).

Additionally, by exploiting the post-condition parametricity of \( R D S T \), we can recover the expressiveness of a 6-place post-condition relation in the \( R S T_0 \) monad that we use in our examples. We show the definition of \( R D S T \) below.

\[
\text{type } \text{RST } \text{pre a post = } \text{RST}_0 \text{a (Ap, } \lambda h : h_0, \text{ pre h_0 h_1 } \\
\land \neg x:0 \ x:1 \ h_0' \ h_1'. \text{ post h_0 h_1 x:0 } x:1 \ h_0' \ h_1' \implies p x:0 x:1 h_0' h_1')
\]

4.3 Lifting classical specifications. To promote reuse of existing verified \( F^* \) code in \( R F^* \), we provide combinators to lift specifications written with classical predicate transformers into the \( R D S T \) monad. To illustrate our approach, we show the \( \lambda_p^\# \) specifications of primitive operations on references—the same combinators apply to arbitrary classically verified code.

\[
\text{type } \text{lift wp0 wp1 } p \ h_0 h_1 = \text{wp0 } (\lambda x:0 h_0'. \wp1 (\lambda x:1 h_1'. \lambda x:0 x:1 h_0' h_1')) h_0
\]

The combinator \( \text{lift} \) takes two classical predicate transformers \( w p_0 \) and \( w p_1 \) and composes them by, in effect, “running” them separately on the heaps \( h_0 \) and \( h_1 \) and relating the results and heaps using the relational post-condition \( p \). The types given to \( \text{deref} \) and assignment should be evident—these are simply the relational liftings of the standard, classical weakest pre-condition rules for these constructs (\( R d \) and \( W \), respectively).

4.4 Computing relational VCs. We repurpose the bulk of \( F^* \)'s type-checking algorithm to \( R F^* \). Although the relational typing rules of Fig. 3 generally analyze a pair of programs \( e_0 \sim e_1 \), for the most part, we are concerned with proving relational properties of multiple executions of a single program. Thus, in the special symmetric case where we are analyzing \( e \sim e \), the two-sided rules of Fig. 3 degenerate into the standard typing rules for monadic \( F^* \) (which is parametric in the choice of monad, so configuring it to use \( R D S T \) is easy).

The main subtlety in computing relational VCs arises when analyzing the cross-cases of conditional expressions—for this we implement the single-sided rules in the judgment, and we attempt to revert to the symmetric case as soon as we detect that the program fragments are indeed the same. For example, the rule \( [\text{if}] \) allows us to relate \( b \) then \( e \) else \( e' \sim e \), by generating subgoals for \( e \sim e \) and \( e' \sim e \), where at least the former can be handled once again by the symmetric rules.

The rules \( \text{[Red-Left]} \) and \( \text{[Red-Right]} \) of Fig. 3 are impossible to implement in full generality—they permit reasoning about stateful programs after arbitrary reductions of open terms. Instead, these rules are approximated by the \( F^* \) typechecker for terms that can be given classical predicate transformer specifications. In particular, when trying to relate \( e_0 \sim e_1 \), if we can use the symmetric judgments and type \( e_0 \sim e_0 : \text{RDST } t \) (\( \text{lift wp}_0 \) \( j \)) and \( e_1 \sim e_1 : \text{RDST } t \) (\( \text{lift wp}_1 \) \( j \)), then we type \( e_0 \sim e_1 \) at type
RDST t (lift wp0 wp1). In effect, by making use of classical predicate transformers on either side, we approximate the reduction relation for stateful terms used by [Red-Left] and [Red-Right].

All these measures for handling the asymmetric cases are still incomplete. When trying to prove a relation between $f v_0 \sim g v_1$ in a context $G$ with relational types for $f$ and $g$ that cannot be decomposed into a pair of classical specifications, it becomes impossible to complete the derivation. In such cases, RF* emits False as the VC (guarded by a relational path condition). Nevertheless, it may still be possible to discharge the VC, if the path condition is feasible. This is the case, for example, when trying to relate the result of encrypt with errorNonce in the passport sample of §2.4.

4.5 Proving VCs using Z3. Once a VC has been computed, we ride on an existing encoding of VCs for the classic Dijkstra monad within Z3. We rely on a theorem from Swamy et al. [46] which guarantees that, despite the use of higher-order logic when computing VCs, once a predicate transformer is applied to a specific first-order post-condition, so long as there is no inherent use of higher-order axioms in the context, a first-order normal form for the VC can always be computed.

5. Applications

Table 5 summarizes our experimental evaluation of RF*. For each program, we give the Makefile target name in the F* distribution, the number of lines of code and type annotations (excluding comments), and the typechecking time in seconds, which is mostly dominated by the time spent solving VCs in Z3. All experiments were conducted on a 3GHz HP Z820 32-core workstation with 32GB of RAM (although the verifier makes use of only one core). For lack of space, most of these examples are only briefly described, with a more detailed discussion of the last two programs, counter in §5.1 (a cryptographic construction) and meter in §5.2 (a privacy protocol).

**INFORMATION FLOW.** The first five programs provide many information flow examples, such as those of §2.2, and test cases for single-sided rules using several variations of the RDST monad construction of §4.

**PASSPORT UNLINKABILITY.** The sixth program, passport illustrates the verification of the Basic Access Control protocol for RFID-equipped passports, presented in §2.4. It establishes passport unlinkability for the modified protocol that returns the same error message in all failure cases. (As can be expected, the original protocol yields a typechecking error.) Its verification illustrates the use of single-sided rules for nested tests (see tag1 in §2.4) and also involves modelling key-hiding symmetric encryption.

**RANDOM ORACLES.** The program rw provides an idealized implementation of a cryptographic hash function in the random oracle model. In this model, widely used in applied cryptography, the hash function is assumed to be indistinguishable from a uniformly random function. Thus, knowledge of the hash function values on a subset of its domain yields no a priori information about its values outside this subset, and protocols that share the hash function with an adversary can treat those values as secret as long as they use disjoint subsets of its domain. The purpose of ro is to capture this reasoning pattern in a library that enables type-based verification of protocols in the random oracle model.

Our implementation lazily samples (and memoizes) the random function, using a usable reference holding a table mapping hash queries made by both honest participants $H$ and adversaries $A$. To verify the program, this table carries several invariants, including that the tables grow monotonically; that in every pair of executions, the tables agree on the fragments corresponding to queries made by $A$; and that on the fragments corresponding to queries by $H$, the sampled entries are related by an injective function that ensures they have indistinguishable distributions. The interface of ro is designed to allow the full use of relational sample on the $H$ fragment, and to account for failure events (e.g., returning a value to $A$ that collides with one that was already provided to $H$), allowing for its modular use in a context that must bound their probability.

**CCA2 ENCRYPTION.** Resistance to adaptive chosen-plaintext and chosen-ciphertext attacks (CCA2) is a standard cryptographic security assumption for public key encryption schemes. To verify protocols relying on this assumption, we program an ideal, stateful functionality for CCA2 encryption that maintains a log of prior oracle communications without disclosing their presence and identities to third parties. Although the protocol has been studied symbolically in the applied pi calculus, to our knowledge we provide its first verification in a computational model of cryptography.

**NONCE-BASED AUTHENTICATION.** Exploiting the modularity of our CCA2 implementation, we program and verify simplenonce, a protocol that illustrates a common authentication pattern based on fresh random values, or nonces, formalizing the intuition that, if $A$ encrypts a fresh nonce using the public key of $B$, and later decrypts a response containing that nonce, then the whole response must have been sent by $B$.

**PRIVATE AUTHENTICATION.** Further extending simplenonce, and relying on a key-hiding variant of CCA2, we implement a protocol for private authentication, proposed by Abadi and Fournet [1], that allows two parties to authenticate one another and to initiate private communications without disclosing their presence and identities to third parties. Although the protocol has been studied symbolically in the applied pi calculus, to our knowledge we provide its first verification in a computational model of cryptography.

**ELGAMAL ENCRYPTION.** The chosen-plaintext security (CPA) of ElGamal encryption by reduction to the decisional Diffie-Hellman assumption is a classic example of cryptographic proof. We verify it in RF*, building on an axiomatic theory of cyclic groups.

**SECURITY “UP TO BAD”.** The program uptobad illustrates a common pattern to prove refinement formulas of the form $\varphi \vee \text{Bad}$ where $\varphi$ is the property we are interested in and $\text{Bad}$ captures conditions that may cause the program to ‘fail’, usually with a small probability (e.g. when the adversary guesses a private key). To
Table 1. Summary of experiments

<table>
<thead>
<tr>
<th>NAME</th>
<th>LoC</th>
<th>TC(S)</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>arith</td>
<td>43</td>
<td>3.7</td>
<td>Information flow with arithmetic</td>
</tr>
<tr>
<td>pure1</td>
<td>35</td>
<td>1.7</td>
<td>Information flow &amp; inference</td>
</tr>
<tr>
<td>pure2</td>
<td>33</td>
<td>1.7</td>
<td>Information flow &amp; inference (variant)</td>
</tr>
<tr>
<td>st</td>
<td>52</td>
<td>3.6</td>
<td>Information flow with state</td>
</tr>
<tr>
<td>singledeside</td>
<td>111</td>
<td>14.2</td>
<td>Information flow using single-sided rules</td>
</tr>
<tr>
<td>passport</td>
<td>97</td>
<td>44.2</td>
<td>Unlinkability for RFID passport protocol</td>
</tr>
<tr>
<td>ro</td>
<td>73</td>
<td>21.2</td>
<td>Random-Oracle hash function</td>
</tr>
<tr>
<td>ccc2</td>
<td>88</td>
<td>6.5</td>
<td>Idealized CCA2 encryption</td>
</tr>
<tr>
<td>simplenocence</td>
<td>108</td>
<td>42.5</td>
<td>Nonce-Based Authentication protocol</td>
</tr>
<tr>
<td>privateauth</td>
<td>175</td>
<td>81.4</td>
<td>Private authentication protocol</td>
</tr>
<tr>
<td>elg</td>
<td>217</td>
<td>124.5</td>
<td>ElGamal encryption</td>
</tr>
<tr>
<td>uptobad</td>
<td>15</td>
<td>1.4</td>
<td>Up-to-failure reasoning</td>
</tr>
<tr>
<td>counter</td>
<td>106</td>
<td>24.1</td>
<td>Counter mode encryptions using AES</td>
</tr>
<tr>
<td>meter</td>
<td>182</td>
<td>79.8</td>
<td>Commitments &amp; smart meter protocol</td>
</tr>
<tr>
<td>Total</td>
<td>1,378</td>
<td>451.3</td>
<td></td>
</tr>
</tbody>
</table>

avoid polluting all our specifications with this disjunction, we define an up-to-bad variant of the RDST monad, where all pre- and post-conditions, and all heap invariants, are enforced only as long as a distinguished boolean memory reference is false. Intuitively, this adds an implicit ‘∧ Bad’ to every refinement.

Our encoding proceeds in two steps. First, we define mref a p, the type of monotonic references r to a value whose contents can be updated only when the update condition p holds. That is, when p is a reflexive transitive binary α-predicate, given two heaps h and h′, if h′ is a successor of h then p (Sel h r) (Sel h′ r) holds. We give below the resulting specification of mwrite, requiring the update condition p as a pre-condition.

```pro
private type mref a p (p: a ⇒ E) = ref a
val mwrite: a; p: a ⇒ E ⇒ r: mref a p ⇒ v: a
  ⇒ RST (λh0 h1. p (Sel h0 (R i)) (L v)) ∧ p (Sel h1 (R i)) (R v)) unit
  (h0 h1 i) h0′ h1′. h0′=Ulpd h0 (L v) (L v)
  ∧ h1=Ulpd (R i) (R v))
```

Next, we define UpTo bad requires a ensures as an alias for the RST monad, with pre-condition requires, result type a, and post-condition ensures, unless the reference bad is set to true in the left or right heap, in which case both pre- and post-conditions are trivial:

```pro
val type UpTo bad a h1 = Sel h0 (L b) b=true ∨ Sel h1 (R b) b=true

val type UpTo (bad: mref bool (λb b′. b′ true⇒ b′ untrue))
  (requires⇒ heap ⇒ a⇒ a ⇒ heap ⇒ heap ⇒ E) = RST (λh0 h1. requires h0 h1. V Bad bad h0 h1)
  a (h0 h1 x0 x1 h0′ h1′. ensures h0 h1 x0 x1 h0′ h1′ ¬ Bad bad h0)
```

Independently, we can compute (or bound) the probability of bad being set to true; for passness, for instance, we set bad to true as we detect a collision between two sampled nonces, and bound its probability with q^2/2^64 where q is the number of the number of experiments.

5.1 Pseudo-random functions and counter-mode. Resuming from the one-time pad example (§2.3), we implement a more useful symmetric encryption scheme based on a block cipher, such as triple-DES or AES. Blocks are just fixed-sized byte arrays, e.g. 16 bytes for AES. Block ciphers take a key and a plaintext block, and produce a ciphertext block. A common cryptographic security assumption is that the block cipher is a pseudo-random function (PRF): for a fixed key, generated uniformly at random, and used only as input to the cipher, the cipher is computationally indistinguishable from a uniformly random function from blocks to blocks. We first present our sample scheme, then formalize the pseudo-random assumption, and finally explain how we verify it by relational typing.

**ENCRYPTION IN COUNTER MODE.** The purpose of symmetric encryption modes is to apply the block cipher keyed with a simple, short secret in order to encrypt many blocks of plaintexts. In counter mode, to encrypt a sequence of plaintext blocks pi, we use a sequence of index blocks li, obtained for instance by incrementing a counter. We independently apply the block cipher to each li to obtain a mask mi: and compute the ciphertext block ci as pi ⊕ mi, effectively using the masks as one-time pads. A practical advantage of this construction is that both encryption and decryption are fully parallelizable, and that the sequence of masks can be pre-computed. The blocks i need not be secret, but they must be pairwise-distinct. Otherwise, from the two ciphertexts p ⊕ m and p′ ⊕ m′, one trivially obtains p ⊕ p′, which leaks a full block of information.

For simplicity, we keep the block cipher key implicit, writing f for the resulting pseudo-random function; we focus on the functions for processing individual plaintext blocks, rather than lists of blocks; and we attach the (public) index to every ciphertext block. First, assume there is a single encryptor, that counts using an integer reference and uses toBytes to format the integer as a block.

```pro
let n = ref 0;
let encrypt (p:block) = let i = toBytes !n in n := !n + 1; (i, xor (f i) p)
let decrypt c i = c xor (f i c)
```

To enable independent encryptions of plaintext blocks, we can remove the global counter and instead sample a block i for each encryption. As follows, this random block i is called the initialization vector (IV) for the encryption.

```pro
let encrypt'(p:block) = let i = sample 16 in (i, xor (f i) p)
```

Much as for the one-time pad, we show that encrypt and encrypt’ can be typed as block → (block × block), the type of functions from (private) blocks to pairs of public blocks, under suitable cryptographic assumptions. More general combinations of sampling and incrementing can also be used for independent multi-block encryptions; for instance, the usual counter mode is programmed as:

```pro
let encrypt_counter_mode (ps: list block) =
  let iv = sample 16 in
  let encrypt (p:block) = let i = ref iv in
    let encrypt (p: int) = let i = sample 16 in
      let encrypt (p: int) = let i = sample 16 in
        (iv, List.map (fun p → incBytes i; xor (f i) p))
  PSEUDO-RANDOM FUNCTIONS. To study the security of protocols using a block cipher, we program and type it as a random function from blocks to blocks. (To test our encryption, we also implement it concretely by just calling AES.) If we can prove the security of a protocol using this ideal random-function implementation, then the same protocol using the concrete block cipher is also secure under the pseudo-random-function assumption (with a probability loss bounded by the probability of distinguishing between the two ciphers). We implement the function f using lazy sampling: when called, f first looks up for a previously-sampled mask in its log; otherwise, f samples a fresh mask. As for the one-time pad, we pass the plaintext block p as a ghost parameter, and take advantage of sampling to generate a mask with a relational refinement to specifically hide p. Of course, this fails if the mask has already been sampled, so we type f for encryption with a pre-condition that depends on the current log and requires that i does not occur in the log yet. (We use the same code with a different type for decryption, requesting that i occurs in the log.)

```pro
val f: index → p:block → RST p:block post block
where
  pre h0 h1 = (Require: i not in the log yet)
  post h0 h1 = (Require: i not in the log yet)
  Mask (p) (R i) m0 m1 (+ and sampled m's related by injectivity)
 ’h0 = Ulpd h0 (L log)’
  ‘h1 = Ulpd h1 (L log)’
  Sampled = Rand h0 h1 (L log)
  let f i p = match assoc i !log with
    | Some (m, i) → m (+ unreachable)
    | None → let m = sample p in
      log := (Entry i m):!log; m
```


When using a single counter (function \textit{encrypt}), typechecking relies on a joint invariant on the counter \( n \) and the content of the log that states that all entries in the log have an index block \( i \) formatted from some \( n < n \). It also involves excluding counter overflows and assuming that \textit{toBytes} is injective. This enables us to prove that our encryption is secure with no loss in the reduction: the advantage of a CPA adversary against our code is the same as the advantage of some adversary against the PRF assumption.

When using instead a fresh random block (function \textit{encrypt}), the situation is more complex, as there is a non-null probability that two different encryptions sample the same index \( i \). Our construction is secure as long as no such collisions happen. We capture this event using the ‘up to bad’ approach presented above, for a Fresh module that silently detects collisions and sets the bad flag accordingly. Concretely, the probability of having a collision when sampling \( q \) blocks of 16 bytes each is bounded by \( q^2 / 2^{256} \).

5.2 Privacy-preserving smart metering & billing. We finally implement and verify the “fast billing” protocol of Rial and Danezis [40], which involves recursive data structures and homomorphic commitments. The protocol has three roles:

- a certified meter that issues private, signed, fine-grained electricity readings (say one reading every 10 minutes);
- a utility company that issues public, signed rates (for the same time intervals, depending on some public policy); and
- a user, who receives both inputs at the end of the month to compute (and presumably pay) his electricity bill.

The two security goals of the protocol are to guarantee (1) integrity of the monthly fee paid to the utility company; and (2) privacy of the detailed readings, which otherwise leak much information on the user’s lifestyle. The protocol relies on Pedersen commitments [36] and public-key signatures. Next, we explain how we prove perfect, information-theoretic privacy (entirely by relational typing) and computational integrity by reduction to the discrete log problem (by typing using ‘up to bad’).

HOMOMORPHIC PEDERSEN COMMITMENTS. We first implement typed commitments, parameterized by some multiplicative group of prime order \( p \). We outline their interface and review their main security properties.

\begin{verbatim}
val pparams = \text{eq public param}
val opening (pp:pparams) (xs:tuple) = openpp (\{ | Eq (trap (L pp + L x) + L o)\})
val sample: ppparams \to \text{list int} \to \{ \text{Readings (fsts xs) \& \text{Commits pp xs}} \}
val commit: ppparams \to \text{list (fsts xrs) \to \{ \text{Commits pp xrs} \}}
val let commit pp x r = pp * g * pp * h * r
val verify pp x r c = (c \text{eq g * x * pp * h * r})
\end{verbatim}

The public parameters \( pp \) consist of the prime \( p \) and two distinct group generators \( g \) and \( h \) (possibly chosen by the utility). Texts and openings range over integers modulo \( p \). A commitment to \( x \) with opening \( o \) is a group element \( c = g^x h^o \). Although assumed hard to compute, there exists \( \alpha \) (unknown to the adversary for these parameters) such that \( g = h^{\alpha} \). Accordingly, we use \( \alpha = \text{trap pp} \) for specification purposes, in refinement formulas but not in the protocol code. We use \( \alpha \) in particular to specify an injective function for randomly sampling the opening \( o \) (modulo \( p \)) so that it perfectly hides \( x \): the relational refinement type opening in the postcondition of \texttt{sample} records that \( \alpha (L x) + (L o) = \alpha (R x) + (R o) \), which implies \( g^{L x} h^{L o} = g^{R x} h^{R o} \) and enables us to type the result of commit as public (eq elt). Intuitively, every commitment can be opened to any \( x' \), for some hard-to-compute \( o' \), so the commitment itself does not leak any information about \( x \) as long as \( o \) is randomly sampled and kept secret. At the same time, given \( x \) and \( o \), it is computationally hard to open the commitment to any \( x' \neq x \).

Commitments can be multiplied: \( g^{h^{\alpha}} h^{\alpha} = g^{h^\alpha} h^{\alpha+o} \) and exponentiated: \( (g^{h^{\alpha}})^{p} = g^{h^{\alpha+p}} \) to compute commitments to linear combinations of their exponents without necessarily knowing them. These operations are used below to compute the bill; their (omitted) types show that they preserve eq and opening relational refinements.

Next, we show some typed code for each role of the protocol.

\textbf{METER}. We have abstract predicates \texttt{Readings} and \texttt{Rates} to specify authentic lists of readings and rates. We rely on a signature scheme to sign a list of commitments to private readings; this scheme is assumed resistant against existential forgery attacks; as explained in [24], we express this property using (non-relational) refinements. For simplicity, we keep the signing and verification keys implicit.

\begin{verbatim}
type Signed (pp:pparams) (cs:cslist elt) =\exists xs. \text{Readings (fsts xs) \& \text{Commits pp xs}}
val sign: ppparams \to csq (list elt) \to \{ \text{Signed pp cs} \} \to \text{eq dsig}
val verify_signature: ppparams \to csq (list elt) \to \text{eq dsig} \to \text{eq bool}
\end{verbatim}

The \texttt{Signed} predicate above states that the commitments have been computed from authentic readings; it is a pre-condition for signing (at the meter) and a post-condition of signature verification (at the utility). It uses a specification function \texttt{fsts} that takes a list of pairs and returns the list of their first projections.

Given authentic readings \( xs \), the \texttt{meter} function below calls \texttt{commits}, a recursive function that maps \texttt{sample} and \texttt{commit} (specified above) to every element of \( xs \) and returns both a list of pairs of readings and commitments \( (x_t, o_t) \) for the user and a public list of commitments \( c_t \) for the utility. These commitments are then signed, yielding a public signature. From their \( eq \) types, we can already conclude that the data passed from the meter to the utility (that is, the list of commitments and its signature) does not convey any information about the readings.

\begin{verbatim}
val meter: pp:pparams \to \text{list int} \to \{ \text{Readings xs} \} \to \text{rates (list (fsts xrs) \& \text{Commits pp xrs})}
val let meter pp xs = let xrs, cs = commits pp xs in (fsts xs, cs, sign pp cs)
\end{verbatim}

\textbf{USER}. Given a list of pairs \( x_t, o_t \) from the meter and a list of rates \( p_t \) from the utility, the user calls \texttt{make_payment} to compute two scalar products: the fee \( \sum x_t p_t \), and a fee opening \( \sum o_t p_t \), and pass them to the utility.

\begin{verbatim}
val make_payment: pp:pparams \to \text{list (fsts xrs) \& \text{Commits pp xrs}} \to \text{rates (list (fsts xrs))} \to \{ \text{Rates (L ps) \& \text{Commits pp xrs}} \to \text{SP (fsts ps L ps) (r ps) \& \text{SP (fsts ps L ps) (r ps) (r ps)}} \to \text{eq bool \& \text{eq bool}}
val let make_payment pp ps xrs = let x,r = sums xrs ps s in (x,r)
\end{verbatim}

The relational pre-condition on the 4th line (SP...) is a declasification condition, capturing the user’s intent to publish the fee, computed as the scalar product \( \text{SP} \) of the detailed readings and rates, by requiring that the ‘left’ and ‘right’ fees be equal. By typing the code of the double scalar product \texttt{sums}, we get the same equation for the openings, showing that the fee opening is then also public. The result type of \texttt{make_payment} tells us that those two scalars reveal no further information on any readings leading to the same fee.

More explicitly, we can use the types of the meter and the user to typecheck a ‘privacy game’ whereby the adversary chooses a list of rates and two lists of readings leading to the same fee; obtains the list of commitments, its signature, the fee, and the fee opening computed by the meter and user code for one of the two readings
each selected at random with probability \( \frac{1}{2} \)); and attempts to guess which of the two readings was used. Typing guarantees that the adversary guess does not depend on the random selection of readings, hence that the guess is correct with probability \( \frac{1}{2} \). Interestingly, this privacy property is information-theoretic, and does not rely on any computational assumption.

**Utility.** The utility verifies the signature on the commitments \( c_i \); uses the rates \( p_s \) to compute the product of exponentials

\[
\prod_i c_i^{p_i} = \prod_i (g^{\pi_i} h^{\mu_i})^{p_i} = \prod_i g^{\pi_i p_i} h^{\mu_i p_i} = g^{\sum_i \pi_i p_i} h^{\sum_i \mu_i p_i}
\]

and compares it to the commitment \( g^r h^o \) computed from the fee \( x \) and fee opening \( o \) presented by the user. Unless the user can open a commitment to several values \( x \) (which can be further reduced to the discrete log problem), this confirms that \( x \) is the correct payment. To type the verifier code, we write classic (but non-trivial) refinements, using ghost scalar products to keep track of its computation.

\[
\text{val verify\_payment: pp:pparams} \\
\quad \rightarrow \text{pseq (list int) [Rates pp]} \\
\quad \rightarrow \text{csseq (list elt) \rightarrow seq disg \{ + from the meter \} } \\
\quad \rightarrow \text{xeq text \rightarrow seq opng \{ + from the user \} } \\
\quad \rightarrow \text{bseq bool \{ btrue \rightarrow 3xs. Readings xs ∧ xeqSP xs\} }
\]

\[
\text{let verify\_payment pp ps cs x r =} \\
\quad \text{verify\_meter\_signature pp cs s ∧} \\
\quad \text{verify\_commit pp x r (scalarExp pp cs ps)}
\]

6. Related work and conclusions

Our work spans semantics of higher-order probabilistic programs, relational program verification, and cryptographic protocol verification, enabling us to verify the security of protocol implementations under computational assumptions by relational typing.

**Reasoning about probabilistic programs.** The semantics of \( \mathsf{RF}^* \) is based on the monadic representation of probabilities used in \([6, 37] \). Our semantics is confined to discrete sub-distributions; for some applications, such as robotics and machine learning, it is however essential to support continuous distributions. Higher-order programs over continuous distributions are considered in \([15, 35] \). An alternative approach is to embed probabilistic programming in a general purpose language, as done e.g. by Kiselyov and Shan \([29] \).

Reif \([38] \), Kozen \([30] \), and Feldman and Harel \([23] \) were among the first to develop logics for reasoning about probabilistic programs. Similar logics were later developed by McIver and Morgan \([33] \) and more recently by Chadha et al. \([17, 18] \). Hurd \([27] \) provides a formalization of the framework of McIver and Morgan in the HOL proof assistant. All these logics are non-relational, and do not allow directly proving relations between probabilities.

**Relational program verification.** Relational Hoare Logic was first introduced for a core imperative program to reason about the correctness of program optimizations and information flow properties \([11] \). It was later extended to probabilistic procedural programs with adversarial code, and used to formally verify reductionist security proofs of cryptographic constructions \([7] \) and differential privacy of randomized algorithms \([10] \). Relational Hoare Type Theory (RHTT) is an extension of Hoare Type Theory, used to reason interactively about advanced information flow policies of higher-order stateful programs with real world data structures \([42] \). RHTT does not consider probabilistic computations, which are essential to reason about cryptographic protocols. RHTT is fully formalized as a shallow embedding in the Coq proof assistant. The formalization is restricted to programs with first-order store; but as we adopt a deep embedding, our formalization could in principle be extended to higher-order store using recent developments in step-indexed semantics \([4] \). Beyond RHTT and HTT, there have been many efforts to develop and sometimes machine-check program logics for higher-order stateful programs; see \([39] \) for an account of the field.

Relational logics can also be used to reason about continuity \([18] \). Naturally, numerous program analyses and specialized relational logics enforce 2-properties of programs.

**Comparison with EasyCrypt.** EasyCrypt \([8] \) is a framework for proving the security of cryptographic constructions in the computational model. The core of EasyCrypt is a probabilistic Relational Hoare Logic (pRHL) that is able to capture common patterns of reasoning in cryptographic proofs, including observational equivalence, equivalence up to failure, and reductionist arguments. The relational refinement type system of \( \mathsf{RF}^* \) is inspired from pRHL, but it also incorporates concerns of compatibility with \( \mathsf{F}^* \) and automation. In contrast to EasyCrypt, \( \mathsf{RF}^* \) offers only limited support to carry relational reasoning about structurally different programs, and to reason about probabilities of postconditions— the latter is achieved in EasyCrypt using a probabilistic (but non-relational) Hoare Logic, which has no counterpart in \( \mathsf{RF}^* \). As a consequence, some cryptographic constructions whose formalization in EasyCrypt requires interactive game-based proofs and complex probabilistic arguments cannot be verified in \( \mathsf{RF}^* \); on the other hand, verification in \( \mathsf{RF}^* \) is fully automatic.

Moreover, EasyCrypt is not primarily designed for building, verifying and deploying large systems. Recent work \([3] \) explores how EasyCrypt can be used to verify C implementations of well-known cryptographic constructions, and through verified compilation derive guarantees about x86 executables. However, this approach does not scale to verifying detailed protocol implementations. In contrast, \( \mathsf{RF}^* \) allows programmers to combine relational and non-relational refinements freely, so that relational reasoning steps can take advantage of program invariants embedded in non-relational refinements. This distinctive ability of \( \mathsf{RF}^* \) is essential to verify system implementations that rely on cryptography.

**Protocol verification.** Blanchet’s recent account of the field of protocol verification provides a panorama of existing tools and major achievements \([14] \). Most of the literature focuses on verifying protocol specifications, or protocol implementations through model extractors \([2] \); alternatives include generating implementations from verified models \([16] \). Our work is most closely related to approaches that reason directly about implementations in the symbolic \([12, 22] \) or computational models \([13, 24, 31] \).

**Modular type-based cryptographic verification \([24] \).** Type systems usefully apply to many notions of security. In the context of computational cryptography, Fournet et al. rely on F7 typed interfaces to encode classic game-based security definitions, such as CPA and CCA2, using a combination of type refinements for authenticity and type abstraction for integrity and confidentiality. In comparison, \( \mathsf{RF}^* \) enables both probabilistic and relational reasoning, letting us typecheck constructions previously out of reach. In their reference implementation of TLS \([13] \), for example, the security of several intricate cryptographic constructions is specified by typing, but justified by handwritten proofs; the corresponding code is trusted, rather than typechecked. (This is the case, e.g. for the MAC-Encode-then-Encrypt construction in the record layer, somewhat similar to our counter mode encryption example in §5.1.) We intend to carry over their 7,000 lines of code to \( \mathsf{RF}^* \), reusing their detailed, classic refinements unchanged, and making use of relational typing to verify additional cryptographic libraries.

**Conclusions and prospects.** Our work on \( \mathsf{RF}^* \) represents a significant first step towards our goal of building software whose
security has been verified down to core cryptographic assumptions. On the theory side, we have shown how to generalize prior probabilistic relational logics to a higher-order language, which provides a formal basis for the use of high-level abstractions in system implementations. Practically, through careful language design, our extension of F∗ towards R F∗ paves the way for carrying out probabilistic relational verification in a semi-automated manner using refinement types and SMT solvers. Still, much remains to be done. As immediate next steps, we anticipate extending our theory to account for dynamic allocation and local state. Toward improving our tools, on the empirical side, we plan to port an existing reference implementation of TLS [13] to R F∗. We expect that R F∗ ’s more flexible idioms, better support for type inference, and its self-certified kernel [43] will ease verification and allow us to push towards obtaining a high-performance implementation of the Internet standard with certified security.

References