Probabilistic Relational Verification for Cryptographic Implementations

Gilles Barthe
IMDEA

Cédric Fournet
Microsoft Research

Benjamin Grégoire
INRIA

Pierre-Yves Strub
MSR-INRIA & IMDEA

Nikhil Swamy
Microsoft Research

Santiago Zanella-Béguelin
Microsoft Research

Abstract

In the form of tools like EasyCrypt, 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. We formalize a core of this logic in Coq and prove its soundness against a denotational semantics for RF*, in contrast to prior operational formalizations of F*. 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.

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 [24], the property underlying information-flow security, relates the observable behaviors of two program executions. In addition to describing multiple executions of a program, security properties must often account for probabilistic behaviors. For instance, simulation- and indistinguishability-based notions of security in cryptography are specificed as a bound on the probability that an adversary wins in some probabilistic experiment.

Recognizing the importance to computer security of such *hyperproperties* [19], 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. In this context, one domain that has seen particularly striking advances is code-based provable security, an emerging approach to cryptography. For example, EasyCrypt [7] is a tool that implements a relational Hoare logic [10] over probabilistic imperative programs. The logic is able to justify common patterns of probabilistic reasoning about hyperproperties used in cryptographic proofs, including observational equivalence, equivalence up to failure, and reductionist arguments. As such, EasyCrypt provides a framework for proving the security of cryptographic constructions in the computational model, which has been used for verifying encryption and signature schemes, modes of operation for block-ciphers, and hash function designs.

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. Still, state-of-the-art formalisms and tools do not quite suffice to achieve this goal. For example, EasyCrypt is not designed for building, verifying and deploying systems. Specifically, the input language of EasyCrypt is an extension of a simple, imperative WHILE language with random assignments and procedure calls, and does not provide the commodities of a general-purpose programming language for developing realistic implementations. More fundamentally, the current version of EasyCrypt does not provide modular reasoning techniques that allow probabilistic guarantees to be lifted to more abstract specifications, so that they can be reused elsewhere in program analyses. Although on-going work aims at extending EasyCrypt with a module system, its purpose is to support composing proofs of cryptographic primitives, rather than scaling up program verification to programs that use them.

On the other hand, another strand of research into code-based security is making fast progress. Several groups develop program verification systems for general purpose languages (ranging from C to ML) and apply them to cryptographic implementations [11, 12, 21, 23, 43]. Based on relatively coarse, non-relational assumptions on cryptographic libraries, these systems have been used to carry out automated, modular proofs (to varying extents) on implementations at a large scale. For example, over the last two years, F7 [11], has been used to verify a 7,000 line, full-fledged implementation [12] of the Transport Layer Security Internet standard (TLS), down to a dozen of computational cryptographic assumptions; and F* [43], another dependently-typed dialect of ML, has been used to verify over 50,000 lines of code, including implementations of multi-party sessions, web-browser extensions, zero-knowledge protocols, and the F* typechecker itself.

RF**: end-to-end security of cryptographic implementations* In an effort to scale code-based security towards end-to-end security proofs of system implementations, this article presents a new language, RF*, which 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 completing the certification of security proofs of critical pieces of Internet infrastructure, such as a reference implementation of TLS [12], using RF*. Technically, this paper makes three broad contributions, discussed next.

1. *A relational logic for higher-order stateful probabilistic programs:* using the Coq proof assistant, we formalize λ_p, a lambda calculus with references, random sampling, and unbounded recursion. We give it a denotational semantics, including both a standard set-theoretical interpretation, as well as a probabilistic, relational interpretation over pairs of store-passing functions. We develop a type system for λ_p and prove it sound with respect to the interpretation. This type system forms a logic for λ_p, the first such logic for higher-order, stateful probablistic programs. (§3)
2. *The design and implementation of* RF**:* the logic of λ_p forms the basis of the design of RF*, an extension of F*. We show how to encode the relational types of λ_p within a new relational state monad, RDST, encoded in F*. We provide a type inference algorithm for RDST in the form a weakest pre-condition calculus that computes relational verification conditions. Proofs of these verification conditions can be discharged automatically by the RF* typechecker and the Z3 [20] SMT solver. (§4)
3. *An experimental evaluation of* RF**:* we demonstrate the expressiveness of RF* through a representative set of examples, starting from simple (non-probabilistic) information flow, and gradually moving towards advanced cryptographic models and systems. To date, we have used 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 RF*, emphasizing the necessity of the λ_p logic for practical security verification. (§2 and §5)

The λ_p theory formalized in Coq, the latest version of the F* compiler with support for relational verification, and all the example programs mentioned in this paper are available from `http://research.microsoft.com/fstar`.

2. Programming with relational refinements

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

2.1 From classic to relational refinements

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 $\mathsf{x{:}t}\{\phi\}$ where ϕ 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:

```
type nat = n:int { 0 ≤ n }
type mod p = n:nat { n < p }
let p = 97
let n : mod p = 73
```

Typechecking F* programs involves logical proof obligations, which are delegated to the Z3 SMT solver. For instance, to check that n has type mod p, the F* typechecker emits the proof obligation $\mathsf{p} = 97 \Longrightarrow 0$ \leq $73 < \mathsf{p}$, which is easily discharged by Z3. Type safety means that, whenever an expression e with type $\mathsf{x{:}t}\{\phi\}$ reduces to a value v, this value v satisfies the formula $\phi[\mathsf{x} := v]$. The type system provides structural subtyping. For instance, nat is a subtype of int, and mod p is a subtype of mod q when $p \leq q$. These subtyping relations are automatically proved and applied by F*.

Refinements can be combined with dependent function types, written $\mathsf{x{:}t} \to \mathsf{t'}$, where the formal parameter x:t is in scope in the result type t'. We also use dependent pairs, written $\mathsf{x{:}t} * \mathsf{t'}$, where the variable x of the first component is in scope in the type 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):

```
val add: p:nat → x:mod p → y:mod p → z:mod p { z = (x + y) % p }
let add p x y = let s = x + y in if s < p then s else s − p
```

F* also provides primitive support for programming with state. For example, one may write **let** incr i = i := !i + 1. By combining refinements with references, one can express invariants on the program state, e.g., **ref** nat is the type of mutable location containing a non-negative integer. To describe more precise properties of effectful programs, F* provides more advanced mechanisms, including a monadic mode [44], where one can reason about programs using variants of the Hoare state monad of Nanevski et al. [33] together with McCarthy's select/update theory for modeling the heap [31]. For example, one can give incr a specification of the form i:**ref** nat → ST (λh.True) unit (λ h () h'.h'=Upd h i (1 + Sel h i)), where ST pre t post can be understood as the state-passing function type h:heap{pre h} → (x:t ∗ h':heap{post h x h'}) although, in reality, F* provides primitive support for state. That is, the type of incr states a trivial pre-condition on the input heap h, and a post-condition indicating that the final heap h' differs from h at the location i, which is incremented. 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 [44].

RF* extends F* with *relational refinements*: every type can (also) be decorated with a relational formula, placed within braces $\{|\ldots|\}$, that specifies a joint property on pairs of values. Relational formulas can independently refer to the *left* and *right* value of every program variable in scope, using the projections L and R, respectively; projections extend naturally to arbitrary formulas. Intuitively, for deterministic programs, type safety means that, whenever we obtain two results v_L and v_R by evaluating an expression e: $\mathsf{x{:}t}\{|\phi|\}$ in two contexts that provide well-typed substitutions for e's free variables, then the formula $\phi[\mathsf{L\,x} := v_L][\mathsf{R\,x} := v_R]$ is valid. More generally, instead of considering two executions of the same program e, RF* allows proving relations between the results of two programs, i.e., we relate e_0 and e_1 at a (relationally refined) type using $e_0 \sim e_1 : \mathsf{t}$. Indeed, we write $e\ :\ \mathsf{t}$ as a shorthand for $e \sim e : \mathsf{t}$.

We start with a few simple examples. Take the expression e to be $\mathsf{z} - \mathsf{z}$; we can give e the type $\mathsf{x{:}int}\{|\mathsf{L\,x} = \mathsf{R\,x}|\}$, meaning that for any pair of substitutions σ_L and σ_R, evaluating $\sigma_L e$ yields the same result as evaluating $\sigma_R e$. Similarly, we have $\mathsf{x} + \mathsf{x} = 2{*}\mathsf{x} : \mathsf{z{:}int}\ \{|\ \mathsf{L\,x} = \mathsf{R\,x} \Longrightarrow \mathsf{L\,z} = \mathsf{R\,z}\ |\}$, stating a simple equivalence between two integer expressions evaluated with the same value for x.

Relational refinements can also be used to describe properties beyond equivalence. Hence, we can express the type of monotonic integer functions as $\mathsf{x{:}int} \to \mathsf{y{:}int}\ \{|\ \mathsf{L\,x} \leq \mathsf{R\,x} \Longrightarrow \mathsf{L\,y} \leq \mathsf{R\,y}\ |\}$ and the type of k-sensitive integer functions with respect to some metric dist as $\mathsf{x{:}int} \to \mathsf{y{:}int}\ \{|\ \mathsf{dist\,(L\,y)\,(R\,y)} \leq \mathsf{k} * \mathsf{dist\,(L\,x)\,(R\,x)}\ |\}$, and let RF* automatically check (by subtyping) that a function such as **fun** x → k ∗ 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 $\{|\ \mathsf{L}\ \phi \wedge \mathsf{R}\ \phi\ |\}$ that independently specifies *left* and *right* properties. For instance, the type nat above is automatically desugared to n:int $\{|\ 0 \leq \mathsf{L\ n} \wedge 0 \leq \mathsf{R\ n}\ |\}$. Pragmatically, this enables us to mix property refinements and relational refinements in our concrete syntax, and to import any refinement-typed F* library in *relational mode* by applying the encoding. When authoring programs with specific relational properties in mind, one need issue only a single compiler directive (aka a pragma) to switch the verifier to relational mode. We contend that the resulting language, 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 non-interference termination-insensitive information flow control [40]. 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 target equivalences. This section provides several examples.

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

```
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-confidentiality values, also written low a. In contrast, we use the type a (the type of unrelated left and right values) for high-confidentiality 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 information flow 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 last_four : n:hi int → s:string{| (L n = R n) % 10000 ⟹ L s = R s |}
let last_four 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;
- $e \sim e'$:low a, assuming L b = true and R b = false; and
- $e' \sim e$: low a, assuming L b = false and R b = true.

Anticipating on the next section, our proof rules for relating two values v_0 and v_1 are relatively simple. Proving $v_0 \sim v_1$: x:t$\{|\ \phi\ |\}$ involves first proving $v_0 \sim v_1$: t (which for base types involves simply showing that both v_0 and v_1 have type t), and then proving $\phi[\mathsf{L\ x}, \mathsf{R\ x} := v_0, v_1]$. 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) |}
→ (x: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 for instance a program that branches on a confidential value and then performs matching public side-effects:

```
val f: x:ref int → b:bool → RST (λ_ _. True) unit post
        where post h0 h1 _ _ h0' h1' = L x=R x ∧ h0=h1⟹ h0'=h1'
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 f 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 similarly be verified.

```
val g: x:ref int → y:ref int → b:bool → RST (λ_ _. True) unit post
        where post h0 h1 _ _ h0' h1' = L x≠L y ∧ R x≠R y
                                       ⟹ Sel h0' (L y)=Sel h1' (R y)
let g x y b = if b then x := 1; y := 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 for the first time 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 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 Section 5 for a description of how we model chosen-ciphertext security in RF*).

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 $c = k \oplus p$; to decrypt c, compute $p = k \oplus c$.

```
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 p0 or p1 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 let cpa' p = encrypt (sample n) p 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 reasoning 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

```
m:block {| xor (L p) (L m) = xor (R p) (R m) |}
```

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

The relational refinement on the result of sample in RF* is not specific to XOR; more generally, it states that sample returns a pair of values related by *any* given one-to-one function on its range. Intuitively, relational refinements in RF* capture relations between the *distribution over values* generated 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.

In programs that contain sample, the interpretation of assertions in the RST monad turns probabilistic. We formalize this in §3, but provide some intuition for their meaning here. If we can give the type RST (λ_ _. True) t Q to $e_0 \sim e_1$, then our logic guarantees that if Q is an equivalence relation partitioning t in a set of equivalence classes $\overline{S}$, and if running e_0 in a heap h_0 reduces to v_0 and e_1 in h_1 reduces to v_1, then for any $S \in \overline{S}$, the probability that $v_0 \in S$ is equal to the probability that $v_1 \in S$. Similar conclusions can be drawn in general. For example, if Q h0 h1 x0 x1 h0' h1' implies P0 x0 $\Longrightarrow$ P1 x1, then the validity of the above assertion implies that Pr[P0 v_0] $\leq$ Pr[P1 v_1].

To reflect these general properties of uniform sampling, our library provides a polymorphic, typed variant of sample, that takes

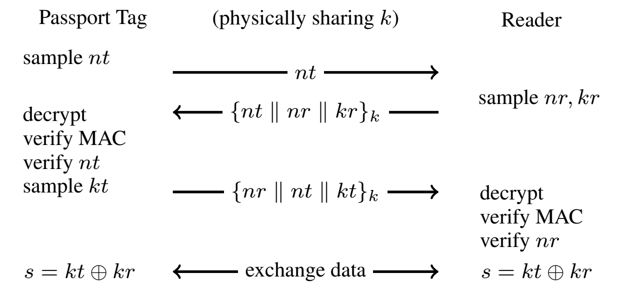

Figure 1. Basic Access Control protocol for passports

as additional ghost parameter F, a binary predicate on sampled values, whose refinement states that it must be an injective function (or, equivalently, a bijection). This parameter has kind A ⇒ A ⇒ E, where E 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

```
type Function F = ∀a.∃ b.F a b ∧ ∀a b1 b2.F a b1 ∧ F a b2 ⟹ b1=b2
type Injective F = Function F ∧
                   ∀a1 a2 b. F a1 b ∧ F a2 b ⟹ a1=a2
val sample: ∀F. len:nat{Injective F} → b:block len{| F (L b) (R b) |}
```

In our one-time pad example, when calling sample n in cpa', we instantiate F to the predicate λ b0 b1. xor (L p) b0 = xor (R p) b1, which is indeed injective. §5 describes security proofs of more realistic encryption 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 protocol for RFID-equipped passports, implemented in RF*, and we discuss a linkability attack against this protocol recently uncovered by Chothia and Smirnov [18]. 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.

Following the ICAO specification for machine-readable travel documents, all recent European passports embed RFID tags featuring the Basic Access Control protocol, outlined in Fig. 1. The protocol has two roles, a passport tag and a reader, exchanging messages using short-distance wireless communications. The goal of the protocol is to establish a shared session key for accessing biometric 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 passports 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 authenticated encryption as triple-DES encryption concatenated with a plaintext Message Authentication Code (MAC). The passport decrypts, 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 it to the concatenation of the two nonces (in a different order than before), and computes the session key $s = kt \oplus kr$. The reader then similarly decrypts the received ciphertext, checks the MAC, and computes s.

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

```
let tag1 k nt c = match decrypt k c with
  | Some p → let (nt',nr,kr) = split p in
               if nt = nt' then encrypt k (concat nr nt (sample_kt()))
               else nonceError
  | 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 error 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 nt and the cipher c with eq refinements (since they are exchanged on a public network), relational typechecking fails on the body of 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 Some p on the left and None on the right, the two resulting expressions are equal, which fails on the proof obligation nonceError = decryptError.

By ensuring that the same error messages are returned in both cases (i.e., by requiring that nonceError = decryptError) this case is prevented, but this alone is not sufficient for verifying the code. Naïvely, 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 $q^2 2^{-64}$ where q is the number of sessions observed by the adversary). We show below the type we assign to tag1—see the RF⋆ program for additional details.

```
val tag1: k:key → nt:nonce → c:cipher → iRST pre block post
where pre h0 h1 = L nt=R nt ∧ L c=R c ∧ (L nt,L k) ∈ Sel h0 (L nts)
                  ∧ (R nt,R k) ∈ Sel h1 (R nts)
  and post h0 h1 b0 b1 h0' h1' = b0=b1
```

The pre-condition of tag1 requires the nonces and ciphertext arguments to be equal (they are sent in the clear). Conversely, the keys may a priori differ. In addition, we maintain ghost state in a reference cell nts that holds a table modeling the association of nonces to keys—the last two clauses of the pre-condition require the nonce/key pairs to be present in this table. As a post-condition, tag1 ensures that the returned blocks, b0 and b1, are equal. In the type, iRST abbreviates the RST monad shown earlier, augmented with a heap invariant that places various constraints on the structure of the nonce table nts and other mutable locations used in the implementation of this program. The iRST monad is the relational analogous of the variant iDST of the Dijkstra monad DST of [44]. Arapinis et al. [4] also analyze the corrected protocol, using the applied π-calculus, essentially proving unlinkability in a more abstract, symbolic model of cryptography.

3. Formal development

We formalize a core of RF⋆ in the Coq proof assistant by developing λ_p, a minimal higher-order language with (statically allocated) references, probabilistic assignments, and unbounded recursion. The formalization is based on the SSREFLECT extension [25], and on the ALEA library for distributions [5]. 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 $G \vdash_{\mathsf{e}} e : \mathbf{T}$ for λ_p. Simple types $\mathbf{T}$ 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 $\mathcal{G} \vdash e_0 \sim e_1 : \mathcal{C}$ that relates a pair of expressions e_0, e_1 in the type $\mathcal{C}$ under the *relational context* $\mathcal{G}$.

We then give a denotational semantics for the introduced type systems. Simple types are given a set-theoretical interpretation $[\![\mathbf{T}]\!]$ in the standard way. Judgments $G \vdash_{\mathsf{e}} e : \mathbf{T}$ are interpreted as the elements of the form $(\!|e|\!)_I$, where I is any valuation for the context G. Taking into account that λ_p is a language with references and probabilistic assignments, the denotation $(\!|e|\!)_I$ of e is defined as a function from memories (equivalently, states or heaps) to distribution over pairs composed of a memory and an element of $[\![\mathbf{T}]\!]$; we denote by $\mathsf{M}([\![\mathbf{T}]\!])$ this function space. Relational types $\mathcal{C}$ are interpreted as a relation $\langle\!|\mathcal{C}|\!\rangle$ over $\mathsf{M}([\![\mathbf{T}]\!])$, where $\mathbf{T}$ is the simple type derived from $\mathcal{C}$ by erasing all refinements. This allows us to interpret (Theorem 1) a valid judgment $\mathcal{G} \vdash e_0 \sim e_1 : \mathcal{C}$ by all the pairs of the form $((\!|e|\!)_{I_\mathcal{L}}, (\!|e|\!)_{I_\mathcal{R}}) \in \langle\!|\mathcal{C}|\!\rangle$ for any pair of valuations $(I_\mathcal{L}, I_\mathcal{R})$ for the erasure G of $\mathcal{G}$.

3.1 λ_p: syntax

λ_p is a simply typed λ-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^j). Formally, the sets of types, contexts, values and expressions are given by the following grammars:

$$
\begin{array}{llll}
\text{type} & \mathbf{T} & ::= & \mathbf{B} \mid \mathbf{T} \to \mathbf{T} \\
\text{ctxt.} & G & ::= & [] \mid G, [x : \mathbf{T}] \\
\text{value} & v, u & ::= & c \mid x \mid o(v_1, \dots, v_n) \mid \mathsf{fun}\ x : \mathbf{T} \to e \\
\text{expr.} & e & ::= & v \mid e\ v \mid\ !r \mid r := v \mid \mathsf{flip} \mid \mathsf{pick}_i^j \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
 & & \mid & \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2
\end{array}
$$

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 $\mathbf{B}$-sorted operators, whose signature is of the form $\mathbf{B}_1 \times \cdots \times \mathbf{B}_n \to \mathbf{B}_0$. We assume that $\mathbf{B}$ contains the unit type (**unit**) along with the types of Booleans (**bool**) and integers (**int**). Their associated constructors are $\bullet$, true, false and $\mathbf{n}$ for $n \in \mathbb{N}$. We implicitly assume that each reference has an ambient base type, and write r : ref $\mathbf{B}$ to denote that r is a reference of type $\mathbf{B}$. The dynamic semantics is defined in the standard way as the compatible closure (for a call-by-value convention) of the $\beta\iota\mu\delta$-contraction.

$$\frac{x : \mathbf{T} \in G \quad \tau \in \{\mathcal{L}, \mathcal{R}\}}{G \mid G' \vdash x_\tau : \mathbf{T}} \qquad \frac{x : \mathbf{T} \in G'}{G \mid G' \vdash x : \mathbf{T}}$$

$$\frac{r : \mathsf{ref}\,\mathbf{B} \quad \tau \in \{\mathcal{L}, \mathcal{R}\}}{G \mid G' \vdash r_\tau : \mathbf{B}} \qquad \frac{G \mid G' \vdash v_i : \mathbf{T}}{G \mid G' \vdash v_1 = v_2 : \mathsf{Prop}}$$

$$\frac{G \mid G', y : \mathbf{B} \vdash \phi : \mathsf{Prop}}{G \mid G' \vdash \forall y : \mathbf{B}.\, \phi : \mathsf{Prop}}$$

Figure 2. Well-formed relational formulas (excerpt)

3.2 λ_p: typing

As usual, a typing context is a sequence of bindings $x : \mathbf{T}$ such that a variable is not bound twice. The typing rules for deriving valid judgments $G \vdash_{\mathsf{v}} v : \mathbf{T}$ and $G \vdash_{\mathsf{e}} e : \mathbf{T}$, in a simply typed setting, are standard and omitted.

RELATIONAL REFINEMENT TYPES. Relational assertions are formulas over tagged variables $x_\mathcal{L}$ or $x_\mathcal{R}$ and tagged references $r_\mathcal{L}$ or $r_\mathcal{R}$; 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 $\mathbf{B}$-sorted formulas built from operators in $\mathcal{O}$ and predicates taken from a set of $\mathbf{B}$-sorted predicates which 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 $\forall y : \mathbf{int}.\, x_\mathcal{L} \leq y \Rightarrow x_\mathcal{R} \leq y + r_\mathcal{R}$ is well-formed under any context G such that $x : \mathbf{int} \in G$, assuming that $r : \mathsf{ref}\,\mathbf{int}$.

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

Refinement types are either *relational types* (denoted by $\mathcal{T}, \mathcal{U}$), 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:

$$\mathcal{T} \;:=\; \mathbf{B} \mid (x : \mathcal{T}) \to \mathcal{C} \qquad \mathcal{C} \;:=\; \{\Phi\} y : \mathcal{T} \{\Psi\}$$

where Φ and Ψ are relational assertions. By convention, x and y are bound in $(x : \mathcal{T}) \to \mathcal{C}$ and $\{\Phi\} y : \mathcal{T}\{\Psi\}$, respectively. However, the type system enforces that x and y only occur in $\mathcal{C}$ and Ψ respectively, if $\mathcal{T}$ is a base type. In other cases, we write $\mathcal{T} \to \mathcal{C}$ and $\{\Phi\}\mathcal{T}\{\Psi\}$.

A *relational context* $\mathcal{G}$ is a sequence of bindings $x : \mathcal{T}$ such that a variable is not bound twice. The refinement type $\mathcal{C}$ is a refinement of T under G, written $G \vdash \mathcal{C} \lhd T$, if T is the result of erasing all pre- and post-conditions occurring in $\mathcal{C}$ and if any assertion that appears in $\mathcal{C}$ is well-formed in G augmented by the local context of Φ in $\mathcal{C}$. This relation is extended to relational contexts: $\mathcal{G} \lhd G$ is the smallest relation such that $[] \lhd []$ and if $\mathcal{G} \lhd G$ and $G \vdash \mathcal{T} \lhd T$ then $\mathcal{G}, x : \mathcal{T} \lhd G, x : T$.

RELATIONAL TYPING. Figure 3 gives a significant subset of the rules defining the relational typing judgments $\mathcal{G} \vdash v_1 \sim v_2 : \mathcal{T}$ and $\mathcal{G} \vdash e_1 \sim e_2 : \mathcal{C}$ for respectively values and expressions. In the figure, $e[x := e_0, e_1]$ stands for $e[x_\mathcal{L} := e_0][x_\mathcal{R} := e_1]$. The full set of rules appear in the accompanying Coq formalization.

A judgment of the form $\mathcal{G} \vdash e_1 \sim e_2 : \{\Phi\}\mathcal{T}\{\Psi\}$ is valid when for any pair of valuations $I_\mathcal{L}, I_\mathcal{R}$ for $\mathcal{G}$ and any pair of *states* m_1, m_2 satisfying the pre-condition Φ, the distributions over values and states obtained by executing e_1 in $I_\mathcal{L}, m_1$ and e_2 in $I_\mathcal{R}, m_2$ are related by the lifting of the post-condition Ψ to distributions.

The formal notion of lifting of a relation to distributions is given below. We anticipate that although assertions in relational refinements are first-order formulae that do not mention probabilities, the definition of lifting is such that valid typing judgments can be used to prove relations between probability quantities. For instance, when Ψ denotes an equivalence relation on $\mathcal{T}$, $[] \models e_1 \sim e_2 : \{\mathsf{true}\}\mathcal{T}\{\Psi\}$ implies that $\Pr[e_1 \in S] = \Pr[e_2 \in S]$, for any equivalence class S of Ψ. Intuitively, observing to which equivalence class the results belong does not help in distinguishing the two expressions.

The rules come in two flavors: double- or single-sided. Double-sided rules allow relating programs with the same head symbol. For instance, rules [LET] and [APP-BASE] are double-sided. They work by relating sub-expressions pairwise, composing pre- and post-conditions using the implicit order of evaluation. Rule [LET] emphasizes this, where the post-condition of the let-bound expression is the pre-condition of the body.

It is not always possible to progress using double-sided rules. For instance, one may want to show that the two expressions (if b then v else v) and v are related by a suitable post-condition. The two expressions having different head symbol, no double-sided rule can apply. Single-sided rules allow to overcome this limitation. The rule [IF-LEFT], which permits to relate an if-expression to an arbitrary expression, is an example of a single-sided rule. All the single-sided rules come in pairs: one variant (tagged [-LEFT]) where the progression is done on the left expression, and one (tagged [-RIGHT]) where the progression is on the right.

Rules for reference assignment ([REF], [REF-LEFT], [REF-RIGHT]), which come in two flavors too, make use of the ability to write assertions about the resulting memory. For example, the two expressions $r := v_0$ and $v := v_1$ are related by a post-condition Ψ when Ψ, where all occurrences of $r_\mathcal{L}$ (resp. $r_\mathcal{R}$) have been replaced by v_0 (resp. v_1), holds as a pre-condition.

Up to now, we only considered rules for expressions headed by non-probabilistic constructions. Rules for random sampling ([FLIP] and [SAMPLE]) are double-sided and require the existence of a bijection f between the support of the two distributions, ensuring a one-to-one correspondence between related values. In the case of flip, we explicitly give the only 2 existing bijections from $\mathbf{bool}$ to $\mathbf{bool}$.

3.3 λ_p: denotational semantics

BACKGROUND. The denotational semantics of well-typed expressions is based on the probability monad [5, 36]. We use the same continuation-passing style formulation as Audebaud and Paulin-Mohring [5].

For the sake of clarity, we only consider distributions over discrete sets. For every discrete set X and $x \in X$, let $\delta_x : X \to [0,1]$ be the Dirac function for x: i.e., $\delta_x(x) = 1$ and $\delta_x(y) = 0$ if $x \neq y$. A sub-distribution over X is a function $\mu : (X \to [0,1]) \to [0,1]$ such that for every $f : X \to [0,1]$, $\mu\ f = \sum_{x \in X} (\mu\ \delta_x)\ (f\ x) \leq 1$. A distribution is a sub-distribution μ such that $\mu\ (\lambda x.\, 1) = 1$. The support $\mathsf{supp}(\mu)$ of a sub-distribution μ is the set of $x \in X$ such that $\mu\ \delta_x > 0$. Every finite set X induces a discrete distribution $\mathcal{U}_X$, that assigns probability $1/|X|$ to each element of X. We let $\mathcal{D}(X)$ be the set of discrete sub-distributions over X. $\mathcal{D}(X)$ inherits from $[0,1]$ the structure of an ω-complete partial order. Moreover, sub-distributions can be given the structure

$$[\textsc{Constr}]\ \frac{G \vdash_{\mathsf{v}} v_0 : \mathbf{B} \qquad G \vdash_{\mathsf{v}} v_1 : \mathbf{B} \qquad \mathcal{G} \lhd G}{\mathcal{G} \vdash v_0 \sim v_1 : \mathbf{B}} \qquad [\textsc{Var}]\ \frac{\mathcal{G} \lhd G \qquad x \in \mathrm{dom}(\mathcal{G})}{\mathcal{G} \vdash x \sim x : \mathcal{G}(x)}$$

$$[\textsc{Fun}]\ \frac{\mathcal{G}, x : \mathbf{B} \vdash e_0 \sim e_1 : \mathcal{C} \qquad \mathcal{G} \lhd G}{\mathcal{G} \vdash \mathsf{fun}\ x : \mathbf{B} \to e_0 \sim \mathsf{fun}\ x : \mathbf{B} \to e_1 : (x : \mathbf{B}) \to \mathcal{C}} \qquad [\textsc{Base-Value}]\ \frac{\mathcal{G} \vdash v_0 \sim v_1 : \mathbf{B}}{\mathcal{G} \vdash v_0 \sim v_1 : \{\Phi[x := v_0, v_1]\} x : \mathbf{B}\{\Phi\}}$$

$$[\textsc{Ref}]\ \frac{r : \mathsf{ref}\,\mathbf{B} \in \mathcal{G} \qquad \mathcal{G} \vdash v_0 \sim v_1 : \mathbf{B}}{\mathcal{G} \vdash r := v_0 \sim r := v_1 : \{\Phi[x := v_0, v_1]\}\mathbf{unit}\{\Phi[x := r]\}} \qquad [\textsc{Ref-Left}]\ \frac{\begin{array}{c} r : \mathsf{ref}\,\mathbf{B} \in \mathcal{G} \qquad \mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{B} \\ \mathcal{G} \vdash e_0 \sim e_1 : \mathcal{C}[x_{\mathcal{L}} := v] \end{array}}{\mathcal{G} \vdash r := v; e_0 \sim e_1 : \mathcal{C}[x_{\mathcal{L}} := r_{\mathcal{L}}]}$$

$$[\textsc{Ref-Right}]\ \frac{\begin{array}{c} r : \mathsf{ref}\,\mathbf{B} \in \mathcal{G} \qquad \mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{B} \\ \mathcal{G} \vdash e_0 \sim e_1 : \mathcal{C}[x_{\mathcal{R}} := v] \end{array}}{\mathcal{G} \vdash e_0 \sim r := v; e_1 : \mathcal{C}[x_{\mathcal{R}} := r_{\mathcal{R}}]} \qquad [\textsc{App-Base}]\ \frac{\mathcal{G} \vdash e_0 \sim e_1 : (x : \mathbf{B}) \to \mathcal{C} \qquad \mathcal{G} \vdash v_0 \sim v_1 : \mathbf{B}}{\mathcal{G} \vdash e_0\ v_0 \sim e_1\ v_1 : \mathcal{C}[x := v_0, v_1]}$$

$$[\textsc{Let}]\ \frac{\mathcal{G} \vdash e_0 \sim e_1 : \{\Phi\}\mathcal{T}\{\Xi\} \qquad \mathcal{G}, x : \mathcal{T} \vdash e_0' \sim e_1' : \{\Xi\}\mathcal{U}\{\Psi\} \qquad x \notin \mathrm{FV}(\mathcal{U}, \Psi)}{\mathcal{G} \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_0' \sim \mathsf{let}\ x = e_1\ \mathsf{in}\ e_1' : \{\Phi\}\mathcal{U}\{\Psi\}}$$

$$[\textsc{Let-Left}]\ \frac{\mathcal{G} \lhd G \qquad G \vdash \mathcal{T} \lhd \mathbf{T} \qquad G \vdash_{\mathsf{e}} e : \mathbf{T} \qquad \mathcal{G}, x : \mathcal{T} \vdash e_0 \sim e_1 : \{\Phi\}\mathcal{U}\{\Psi\} \qquad x \notin \mathrm{FV}(\mathcal{U}, e_1, \Psi)}{\mathcal{G} \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e_0 \sim e_1 : \{\Phi\}\mathcal{U}\{\Psi\}}$$

$$[\textsc{Let-Right}]\ \frac{\mathcal{G} \lhd G \qquad G \vdash \mathcal{T} \lhd \mathbf{T} \qquad G \vdash_{\mathsf{e}} e : \mathbf{T} \qquad \mathcal{G}, x : \mathcal{T} \vdash e_0 \sim e_1 : \{\Phi\}\mathcal{U}\{\Psi\} \qquad x \notin \mathrm{FV}(\mathcal{U}, e_0, \Psi)}{\mathcal{G} \vdash e_0 \sim \mathsf{let}\ x = e\ \mathsf{in}\ e_1 : \{\Phi\}\mathcal{U}\{\Psi\}}$$

$$[\textsc{LetRec}]\ \frac{f, x \notin \mathrm{FV}(\mathcal{V}, \Psi) \qquad \mathcal{G}, f : (x : \mathcal{T}) \to \{\Phi\}\mathcal{U}\{\Phi\} \vdash e_0' \sim e_1' : \{\Xi\}\mathcal{V}\{\Psi\} \qquad \mathcal{G}, f : (x : \mathcal{T}) \to \{\Phi\}\mathcal{U}\{\Phi\}, x : \mathcal{T} \vdash e_0 \sim e_1 : \{\Phi\}\mathcal{U}\{\Phi\}}{\mathcal{G} \vdash \mathsf{letrec}\ f\ x = e_0\ \mathsf{in}\ e_0' \sim \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_1' : \{\Xi\}\mathcal{V}\{\Psi\}}$$

$$[\textsc{If-Left}]\ \frac{\mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{bool} \qquad \mathcal{G} \vdash e_1 \sim e : \{\Phi_1\}\mathcal{C}\{\Psi\} \qquad \mathcal{G} \vdash e_2 \sim e : \{\Phi_2\}\mathcal{C}\{\Psi\}}{\mathcal{G} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \sim e : \{(v_{\mathcal{L}} \Rightarrow \Phi_1) \wedge (\neg v_{\mathcal{L}} \Rightarrow \Phi_2)\}\mathcal{C}\{\Psi\}}$$

$$[\textsc{If-Right}]\ \frac{\mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{bool} \qquad \mathcal{G} \vdash e \sim e_1 : \{\Phi_1\}\mathcal{C}\{\Psi\} \qquad \mathcal{G} \vdash e \sim e_2 : \{\Phi_2\}\mathcal{C}\{\Psi\}}{\mathcal{G} \vdash e \sim \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \{(v_{\mathcal{R}} \Rightarrow \Phi_1) \wedge (\neg v_{\mathcal{R}} \Rightarrow \Phi_2)\}\mathcal{C}\{\Psi\}}$$

$$[\textsc{If}]\ \frac{\mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{bool} \qquad \mathcal{G} \vdash e_1 \sim e_1' : \{\Phi_1\}\mathcal{C}\{\Psi\} \qquad \mathcal{G} \vdash e_2 \sim e_2' : \{\Phi_2\}\mathcal{C}\{\Psi\}}{\mathcal{G} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \sim \mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2' : \{(v_{\mathcal{L}} \iff v_{\mathcal{R}}) \wedge (v_{\mathcal{L}} \Rightarrow \Phi_1) \wedge (\neg v_{\mathcal{L}} \Rightarrow \Phi_2)\}\mathcal{C}\{\Psi\}}$$

$$[\textsc{Flip}]\ \frac{f = x \mapsto x \text{ or } f = x \mapsto \neg x}{\mathcal{G} \vdash \mathsf{flip} \sim \mathsf{flip} : \{\forall y : \mathbf{bool}.\, \Phi[x := y, f(y)]\}x : \mathbf{bool}\{\Phi\}} \qquad [\textsc{Sample}]\ \frac{f \in \mathbb{N} \to \mathbb{N}, \text{bijective from } [i..j] \text{ to } [i..j]}{\mathcal{G} \vdash \mathsf{pick}_i^j \sim \mathsf{pick}_i^j : \{\Phi[x := y, f(y)]\}x : \mathbf{int}\{\Phi\}}$$

$$[\textsc{Red-Left}]\ \frac{e_1 \xrightarrow{\beta\iota\mu\delta} e_2 \qquad \mathcal{G} \vdash e_1 \sim e : \mathcal{C}}{\mathcal{G} \vdash e_2 \sim e : \mathcal{C}} \qquad [\textsc{Red-Right}]\ \frac{e_1 \xrightarrow{\beta\iota\mu\delta} e_2 \qquad \mathcal{G} \vdash e \sim e_1 : \mathcal{C}}{\mathcal{G} \vdash e \sim e_2 : \mathcal{C}}$$

Figure 3. Relational typing rules

of a monad, by taking as unit and composition operators:

$$\begin{aligned} \mathsf{unit} &: X \to \mathcal{D}(X) \triangleq \lambda x\, f.\, f\, x \\ \mathsf{bind} &: \mathcal{D}(X) \to (X \to \mathcal{D}(Y)) \to \mathcal{D}(Y) \\ &\triangleq \lambda \mu\, M\, f.\, \mu\, (\lambda\, x. M\, x\, f) \end{aligned}$$

The interpretation of relational types rests on an operator $\cdot^\sharp$ that lifts relations over $A \times B$ into relations over $\mathcal{D}(A) \times \mathcal{D}(B)$; the operator is inspired from early works on probabilistic bisimulations [27], and is used in CertiCrypt [6] and EasyCrypt [8] to interpret relational judgments. Formally, let $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$; then $R^\sharp\ \mu_1\ \mu_2$ iff:

$$\exists \mu : \mathcal{D}(A \times B).\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{supp}(\mu) \subseteq R$$

where π_1 and π_2 are the projections for distributions over pairs, i.e.

$$\begin{aligned} \pi_1(\mu) &= \mathsf{bind}\, \mu\ (\lambda(x, y).\, \mathsf{unit}\ x) \\ \pi_2(\mu) &= \mathsf{bind}\, \mu\ (\lambda(x, y).\, \mathsf{unit}\ y) \end{aligned}$$

A fundamental property of this lifting operator is that given $f : A \to [0, 1]$ and $g : B \to [0, 1]$ such that

$$\forall a \in A, b \in B.\ R\ a\ b \Rightarrow f\ a = g\ b$$

then $R^\sharp\ \mu_1\ \mu_2$ implies that $\mu_1\ f = \mu_2\ g$, i.e. the expected values of f in μ_1 and g in μ_2 coincide. Note that events can be viewed as $\{0, 1\}$-valued functions, and in this case expectation coincides with probability.

SET-THEORETICAL INTERPRETATION. We assume each base type $\mathbf{B}$ is given a set-theoretical interpretation $[\![B]\!]$, and that each constructor c belonging to the base type $\mathbf{B}$ is given a denotation $\mathcal{B}(c) \in [\![B]\!]$. We define the set of semantical values as $\bigcup_{\mathbf{B}} [\![B]\!]$, and then the set $\mathcal{M}$ of *states* as the set of well-typed mappings from references to semantical values.

$$\mathcal{M} = \{m : \mathsf{ref} \to \bigcup_{\mathbf{B}} [\![B]\!] \mid \forall r : \mathsf{ref}\,\mathbf{B}.\, m(r) \in [\![B]\!]\}$$

$$
\begin{array}{ll}
\llparenthesis c \rrparenthesis_I & = \mathcal{B}(c) \\
\llparenthesis x \rrparenthesis_I & = I(x) \\
\llparenthesis \mathsf{fun}\ x : \mathbf{T} \to e \rrparenthesis_I & = \lambda d.\, \lambda m.\, \llparenthesis e \rrparenthesis^m_{I[x:=d]} \\
\\
\llparenthesis v \rrparenthesis^m_I & = \mathsf{unit}\ (\llparenthesis v \rrparenthesis_I, m) \\
\llparenthesis e\ v \rrparenthesis^m_I & = \mathsf{bind}\ \llparenthesis e \rrparenthesis^m_I\ (\lambda f.\, \lambda m.\, (f\ \llparenthesis v \rrparenthesis_I m, m)) \\
\llparenthesis \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rrparenthesis^m_I & = \mathsf{bind}\ \llparenthesis e_1 \rrparenthesis^m_I\ (\lambda d.\, \lambda m.\, (\llparenthesis e_2 \rrparenthesis^m_{I[x:=d]}, m)) \\
\llparenthesis \mathsf{letrec}\ f\ x = e_1\ \mathsf{in}\ e_2 \rrparenthesis^m_I & \\
 & = \mathsf{bind}\ (\mathrm{lub}\ F_{fx \mapsto e_1})\ (\lambda d.\, \lambda m.\, \llparenthesis e_2 \rrparenthesis^m_{I[f:=d]}) \\
\llparenthesis !r \rrparenthesis^m_I & = \mathsf{unit}\ (m(r), m) \\
\llparenthesis r := e \rrparenthesis^m_I & = \mathsf{bind}\ \llparenthesis e \rrparenthesis^m_I\ (\lambda d.\, \lambda m.\, (\bullet, m[r := d])) \\
\llparenthesis \mathsf{flip} \rrparenthesis^m_I & = \mathsf{bind}\ \mathcal{U}_{\mathcal{B}}\ (\lambda b.\, (b, m)) \\
\llparenthesis \mathsf{pick}^j_i \rrparenthesis^m_I & = \mathsf{bind}\ \mathcal{U}_{[i,j]}\ (\lambda n.\, (n, m))
\end{array}
$$

where $F_{fx \mapsto M}$ is defined as

$$
\begin{array}{l}
F_{fx \mapsto M} : 0 \qquad \mapsto \mathsf{unit}\ (\lambda d_x.\, \lambda s.\, \lambda g.\, 0) \\
F_{fx \mapsto M} : n+1 \mapsto \\
\quad \mathsf{bind}\ (F_{fx \mapsto M}(n))(\lambda d_f.\, \lambda d_x.\, \lambda s.\, \llparenthesis M \rrparenthesis^s_{I[f:=d_f][x:=d_x]})
\end{array}
$$

Figure 4. Interpretation of values and expressions

Then we extend the interpretation to functional types by setting

$$[\![T_1 \to T_2]\!] = [\![T_1]\!] \to \mathsf{M}([\![T_2]\!])$$

where $\mathsf{M}(X) \stackrel{\triangle}{=} \mathcal{M} \to \mathcal{D}(\mathcal{M} \times X)$.

A valuation I is a function that maps every declaration $x : \mathbf{T}$ to a semantical value. A valuation I is well-formed for G, written $I \vDash G$, if I maps every declaration $x : \mathbf{T}$ in G to an element of $[\![\mathbf{T}]\!]$. Let I be a valuation and m be a memory. The interpretations $\llparenthesis v \rrparenthesis_I$ of a value v and $\llparenthesis e \rrparenthesis^m_I$ of an expression e are defined in Figure 4. If I is a well-formed valuation for G, and $G \vdash_{\mathsf{v}} v : \mathbf{T}$ is derivable, then $\llparenthesis v \rrparenthesis_I \in [\![\mathbf{T}]\!]$. Likewise, if $G \vdash_{\mathsf{e}} e : \mathbf{T}$ is derivable then $\lambda m.\, \llparenthesis e \rrparenthesis^m_I \in \mathsf{M}([\![\mathbf{T}]\!])$.

RELATIONAL INTERPRETATION. A well-formed *relational valuation* $\mathcal{I}$ for G, written $\mathcal{I} \vDash G$, is a pair of well-formed valuations for G. If $\mathcal{I} = (I_{\mathcal{L}}, I_{\mathcal{R}})$, we write $\pi_1(\mathcal{I})$ (resp. $\pi_2(\mathcal{I})$) for $I_{\mathcal{L}}$ (resp. $I_{\mathcal{R}}$), and $\mathcal{I}(x)$ for $(I_{\mathcal{L}}(x), I_{\mathcal{R}}(x))$. We assume given a relational interpretation for formulas, written $\langle\!| \Phi |\!\rangle_{\mathcal{I}}$, such that for any formula Φ well-formed under G, for any relation valuation $\mathcal{I} \vDash F$, $\langle\!| \Phi |\!\rangle_{\mathcal{I}}$ is a relation on $\mathcal{M}$. This relation is defined as usual, using the left-/right valuation (resp. left/right memory argument) for interpreting variables on the left/right (resp. references on the left/right).

Figure 5 defines the interpretation of relational types, written $\langle\!| G \vdash \mathcal{T} \triangleleft \mathbf{T} |\!\rangle_{\mathcal{I}}$, and computation types, written $\langle\!| G \vdash \mathcal{C} \triangleleft \mathbf{T} |\!\rangle_{\mathcal{I}}$, w.r.t. a relational valuation $\mathcal{I}$. A relational valuation $\mathcal{I}$ is well-formed w.r.t a relational context $\mathcal{G}$, written $\mathcal{I} \vDash \mathcal{G}$, if for any context G such that $\mathcal{G} \triangleleft G$, and every variable x declared in G, $\mathcal{I}(x) \in \langle\!| G \vdash \mathcal{G}(x) \triangleleft G(x) |\!\rangle_{\mathcal{I}}$.

Finally, we define the semantic validity of judgments: we say that two values v_1 and v_2 are semantically related in $\mathcal{T}$ under $\mathcal{G}$, written $\mathcal{G} \vDash v_1 \sim v_2 : \mathcal{T}$, if $\mathcal{G} \triangleleft G$, and $G \vdash \mathcal{T} \triangleleft T$, and

$$\forall \mathcal{I} \vDash \mathcal{G},\ (\llparenthesis v_1 \rrparenthesis_{\pi_1(\mathcal{I})}, \llparenthesis v_2 \rrparenthesis_{\pi_2(\mathcal{I})}) \in \langle\!| G \vdash \mathcal{T} \triangleleft \mathbf{T} |\!\rangle_{\mathcal{I}}$$

We say that two expressions e_1 and e_2 are semantically related in $\mathcal{C}$ under $\mathcal{G}$, written $\mathcal{G} \vDash e_1 \sim e_2 : \mathcal{C}$, if $\mathcal{G} \triangleleft G$, and $G \vdash \mathcal{T} \triangleleft T$, and

$$\forall \mathcal{I} \vDash \mathcal{G},\ (\lambda m.\, \llparenthesis e_1 \rrparenthesis^m_{\pi_1(\mathcal{I})}, \lambda m.\, \llparenthesis e_2 \rrparenthesis^m_{\pi_2(\mathcal{I})}) \in \langle\!| G \vdash \mathcal{C} \triangleleft \mathbf{T} |\!\rangle_{\mathcal{I}}$$

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 $\mathcal{G} \vdash v_1 \sim v_2 : \mathcal{T}$, then $\mathcal{G} \vDash v_1 \sim v_2 : \mathcal{T}$,*
- *If $\mathcal{G} \vdash e_1 \sim e_2 : \mathcal{C}$, then $\mathcal{G} \vDash e_1 \sim e_2 : \mathcal{C}$.*

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 λ_p in RF*

We now discuss our language design and implementation. There are two key ideas behind our encoding of λ_p in RF*. 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 ST to a monad RST 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 RST monad, and the manner in which we reuse classical specifications.

4.1 Representing λ_p types

To implement λ_p, we begin with a translation of its types into F* augmented with a relational state monad. To stay close to λ_p, our translation uses a monad RST0, which we then adapt to the monad RST of §2. Like in λ_p, post-conditions in RST0 only relate the output values and heaps, not the initial heaps. Specifically, the type RST0 pre a post can be interpreted in RF* as a store-passing function (over a primitive heap) with the signature shown below:

```
RST0 pre a post = h:heap{|pre (L h) (R h)|}
                  → (x:a ∗ h':heap{| post (L x) (R x) (L h') (R h') |})
```

The type translation is homomorphic on most of λ_p's typing constructs, with the interesting cases mainly on the computation types. For first-order computation types $[\![\{\Phi\}\mathbf{B}\{\Psi\}]\!]$ is RST0 $[\![\Phi]\!]$ $[\![\mathbf{B}]\!]$ $[\![\Psi]\!]$, while, higher-order computation types in λ_p, $[\![\{\Phi\}\mathcal{U}\{\Psi\}]\!]$ (whose post-condition is not dependent on the result $\mathcal{U}$), are represented as RST0 $[\![\Phi]\!]$ $[\![\mathcal{U}]\!]$ (λ_ _.$[\![\Psi]\!]$).[1]

4.2 A monad of predicate transformers for VC generation

Next, to provide type inference for RF*, rather than writing relational Hoare triples in RST0, we write specifications using predicate transformers. This style is adapted from the *Dijkstra state monad*, previously introduced for inferring classical (non-relational) verification conditions for stateful F* programs [44]. In particular, we introduce the *relational* Dijkstra state monad, RDST, and show its signature below. (Note, we write polymorphic types implicitly assuming their free type variables are prenex quantified.)

```
type RDST a wp = ∀p. RST0 (wp p) a p
val return : x:a → RDST a (Λp. p (L x) (R x))
val bind: RDST a wp1 → (x:a → RDST b (wp2 x))
  → RDST b (Λp.λh0 h1. wp1 (λx0 x1 h0' h1'.
                 (∀ x. L x=x0 ∧ R x=x1 ⟹ wp2 x p h0' h1')) h0 h1)
```

The type RDST t wp is an abbreviation for the RST0 monad that is polymorphic in its post-condition. Specifically, RDST t wp is the type of computation which for any relational post-condition p on ts and heaps, the pre-condition on the input heaps is given by wp p.

Unlike the Hoare-style RST0 monad, the RDST monad yields a weakest pre-condition calculus by construction. As indicated by the signature of bind, when composing computations in the RDST monad, we simply compute a pre-condition for the computation by composing the predicate transformers of each component. A slight complication arises from the need to constrain the formal parameter

[1] In principle, in RF*, one could write types like RST0 p(x:t → RST0 p' t' q') (λf0 f1. q) where q mentions f0, f1, which is inexpressible in λ_p. However, such a type is generally useless in RF*, since the functions f0, f1 cannot be applied in q.

$$\frac{f_1, f_2 \in [\![\mathbf{T}]\!] \to \mathcal{M} \to \mathcal{D}([\![\mathbf{U}]\!] \times \mathcal{M}) \qquad \forall t_1, t_2 \in (\!| G \vdash \mathcal{T} \lhd \mathbf{T} |\!)_{\mathcal{I}}.\, (f_1\ t_1, f_2\ t_2) \in (\!| G \vdash \mathcal{C} \lhd \mathbf{U} |\!)_{I[x:=(t_1,t_2)]}}{(f_1, f_2) \in (\!| G \vdash (x : \mathcal{T}) \to \mathcal{C} \lhd \mathbf{T} \to \mathbf{U} |\!)_{\mathcal{I}}}$$

$$\frac{(d_1, d_2) \in [\![\mathbf{B}]\!]^2}{(d_1, d_2) \in (\!| G \vdash \mathbf{B} \lhd \mathbf{B} |\!)_{\mathcal{I}}} \qquad \frac{\mu_1, \mu_2 \in \mathcal{M} \to \mathcal{D}([\![\mathbf{U}]\!] \times \mathcal{M}) \qquad \forall m_1, m_2 \in \mathcal{M}.\, (\!|\Phi|\!)_{\mathcal{I}}(m_1, m_2) \Rightarrow P^{\sharp}\ (\mu_1\ m_1)\ (\mu_2\ m_2)}{(\mu_1, \mu_2) \in (\!| G \vdash \{\Phi\} y : \mathcal{U}\{\Psi\} \lhd \mathbf{U} |\!)_{\mathcal{I}}}$$

where $P = \lambda((u_1, m_1'), (u_2, m_2')).\, (u_1, u_2) \in (\!| G \vdash \mathcal{U} \lhd \mathbf{U} |\!)_{\mathcal{I}} \wedge (\!|\Psi|\!)_{\mathcal{I}[y:=(u_1,u_2)]}(m_1', m_2')$

Figure 5. Interpretation of relational refinement types

x:a of wp2 relationally. In general, wp2 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=x0 and R x=x1, before composing wp1 and wp2.

Additionally, by exploiting the post-condition parametricity of RDST, we can recover the expressiveness of a 6-place post-condition relation in the RST monad that we use in our examples. We show the definition of RST below.

```
type RST pre a post = RDST a (Λp.λh0 h1. pre h0 h1
        ∧ ∀x0 x1 h0' h1'. post h0 h1 x0 x1 h0' h1' ⟹ p x0 x1 h0' h1')
```

4.3 Lifting classical specifications

To promote reuse of existing verified F⋆ code in RF⋆, we provide combinators to lift specifications written with classical predicate transformers into the RDST monad. To illustrate, we show the RF⋆ specifications of primitive operations on references—the same combinators apply to arbitrary classically verified code.

```
type lift wp0 wp1 p h0 h1 =
      wp0 (λx0 h0'. wp1 (λx1 h1'. p x0 x1 h0' h1') h1) h0

type Rd x p h = p (Sel h x) h
val (!) : x:ref a → RDST a (lift (Rd (L x)) (Rd (R x)))

type W x v p h = p () (Upd h x v)
val (:=): x:ref 'a→ v:'a→ RDST unit (lift (W (L x) (L v)) (W (R x) (R v)))
```

The combinator lift takes two classical predicate transformers wp0 and wp1 and composes them by, in effect, "running" them separately on the heaps h0 and h1 and relating the results and heaps using the relational post-condition p. The types given to dereference and assignment should be evident—these are simply the relational liftings of the standard, classical weakest pre-condition rules for these constructs (Rd and W, respectively).

4.4 Computing relational VCs

We repurpose the bulk of typechecking of F⋆ to RF⋆. 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 RDST 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 attempt to revert to the symmetric case as soon as we detect that the program fragments are indeed the same. For example, the rule [IF] allows us to relate **if** 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 rule [RED-LEFT] of Fig. 3 is impossible to implement in full generality—it permits reasoning about stateful programs after arbitrary reductions of open terms. However, the rule is approximated by the RF⋆ 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 $e_0 \sim e_0$: RDST t (lift wp0 _), and $e_1 \sim e_1$: RDST t (lift _ wp1), 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 its symmetric counterpart).

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 $\mathcal{G}$ 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 infeasible. This is the case, for example, when trying to relate the result of encrypt with errorNonce in the passport example of §2.4.

4.5 Proving VCs using Z3

Once a VC has been computed, we ride on an existing encoding of VCs computed for the classic Dijkstra monad within Z3. We rely on a theorem from Swamy et al. [44] 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 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. 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.

NAME	LOC	TC(S)	DESCRIPTION
arith	43	4.5	Information flow with arithmetic
pure1	35	1.7	Information flow & inference
pure2	33	1.7	Information flow & inference (variant)
st	52	3.6	Information flow with state
singlesided	111	14.2	Information flow using single-sided rules
passport	97	44.2	Unlinkability for RFID passport protocol
ro	73	21.2	Random-Oracle hash function
cca2	88	6.5	Idealized CCA2 encryption
simplenonce	108	42.5	Nonce-based Authentication protocol
privateauth	175	81.4	Private authentication protocol
elg	217	124.5	ElGamal encryption
uptobad	15	1.4	Up-to-failure reasoning
counter	106	24.1	Counter mode encryptions using AES
meter	182	79.8	Commitments & smart meter protocol
Total	1,378	451.3	

Table 1. Summary of experiments

RANDOM ORACLES. The program ro 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 mutable 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 encryptions, similar to those proposed by Fournet et al. [23], but with a more convenient relational interface. Using the F7 type system with only classic refinements, they require that all code that operates on secrets be placed in a separate module that exports plaintexts as an abstract type. Using instead relational types for secrets, in the style of §2.3, we lift this restriction, enabling us to verify protocol code that uses encryption without restructuring. Our code is essentially higher-order; it simulates ML functors using a dependently typed record of functions.

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 $\varphi \vee$ Bad where φ is the property we are interested in and 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 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 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 an a-value whose contents can be updated only when the update condition p holds. That is, when p is a reflexive transitive binary a-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.

```
(* monotonic reference, an abstract alias for a reference *)
private type mref a (p:a ⇒ a ⇒ E ) = ref a
val mwrite: a::⋆ → p::a⇒ a⇒ E → r:mref a p → v:'a
   → RST (λh0 h1. p (Sel h0 (L r)) (L v) ∧ p (Sel h1 (R r)) (R v)) unit
         (λh0 h1 () () h0' h1'. h0'=Upd h0 (L r) (L v)
                                ∧ h1'=Upd h1 (R r) (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:

```
type Bad b h0 h1 = Sel h0 (L b)=true ∨ Sel h1 (R b)=true
type UpTo (bad:mref bool (λb b'. b=true ⟹ b'=true))
          (requires:heap ⇒ heap ⇒ E) (a::⋆ )
          (ensures::heap ⇒ heap ⇒ a ⇒ a ⇒ heap ⇒ heap ⇒ E) =
   RST (λh0 h1. requires h0 h1 ∨ Bad bad h0 h1) a
       (λh0 h1 x0 x1 h0' h1'. ensures h0 h1 x0 x1 h0' h1'
       ∨ Bad bad h0 h1)
```

Independently, we can compute (or bound) the probability of bad being set to true; for passport, 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 sessions.

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.

ENCRYPTING IN COUNTER MODE. The purpose of symmetric encryption modes is to apply the block cipher keyed with a single, short secret in order to encrypt many blocks of plaintexts. In counter mode, to encrypt a sequence of plaintext blocks p_i, we use a sequence of index blocks i_i, obtained for instance by incrementing a counter. We independently apply the block cipher to each i_i to obtain a mask m_i; and compute the ciphertext block c_i as $\mathsf{p}_i \oplus \mathsf{m}_i$, 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 $\mathsf{p} \oplus \mathsf{m}_i$ and $\mathsf{p}' \oplus \mathsf{m}_i$, one trivially obtains $\mathsf{p} \oplus \mathsf{p}'$, 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.

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

```
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 → eq (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:

```
let encrypt_counter_mode (ps:list block) =
  let iv = sample 16 in let i = ref iv in
  iv::List.map (fun p → incrBytes i; xor (f !i)) ps
```

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.)

```
val f: i:index → p:block → iRST pre block post
where pre h0 h1 = (∗ Requires: i not in the log yet ∗)
      not (In (L i) (Domain (Sel h0 (L log))))
    ∧ (L i = R i) ∧ (Seqn (L i) < Sel h0 (L n))
  and post h0 h1 m0 m1 h0' h1' = (∗ Ensures: log extended with (i,p,m) ∗)
      Mask (L p) (R p) m0 m1 (∗ and sampled m's related by injectivity ∗)
    ∧ h0' = Upd h0 (L log) ((Entry (L i) (L p) m0)::Sel h0 (L log))
    ∧ h1' = Upd h1 (R log) ((Entry (R i) (R p) m1)::Sel h1 (R log))
let f p i = match assoc i !log with
  | Some(_,m) → m (∗ unreachable ∗)
  | None → let m = sample p in
           log := (Entry i p m)::!log;
           m
```

When using a single counter (function 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 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 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 $q^2/2^{128}$. By typing, we prove that encryption, and any program that may use it, leaks information only once *bad* is true. Thus, we prove the concrete security of encrypt' with a loss of $q^2/2^{128}$ in the reduction to PRF.

5.2 Privacy-preserving smart metering & billing

We finally implement and verify the "fast billing" protocol of Rial and Danezis [39], 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 [35] 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 (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.

```
type pparams = eq public_param
type opening (pp:pparams) (x:text) =
  o:opng {| Eq ((trap (L pp) ∗ L x) + L o) ((trap (R pp) ∗ R x) + R o) |}

val sample: pp:pparams → x:text → opening pp x
val commit: pp:pparams → x:text → r:opening pp x →
  c:eq elt { c = Commit pp x r }
let commit pp x r = pp.g ˆ x ∗ pp.h ˆ r
let verify pp x r c = (c = pp.g ˆ x ∗ pp.h ˆ r)
```

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 α (known as the *trapdoor* for these parameters) such that $g = h^\alpha$. Accordingly, we use α = trap pp for specification purposes, in refinement formulas but not in the protocol code. We use α 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 post-condition of sample records that $\alpha{*}(\mathsf{L}\ x) + (\mathsf{L}\ o) = \alpha{*}(\mathsf{R}\ x) + (\mathsf{R}\ o)$, which implies $g^{Lx}h^{Lo} = g^{Rx}h^{Ro}$ 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: $g^x h^o * g^{x'} h^{o'} = g^{x+x'} h^{o+o'}$ and exponentiated: $(g^x h^o)^p = g^{xp} h^{op}$ 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.

METER. We have abstract predicates Readings and 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 [23], we express this property using (non-relational) refinements. For simplicity, we keep the signing and verification keys implicit.

```
type Signed (pp:pparams) (cs:list elt) =
    ∃xrs. Readings (fsts xrs) ∧ cs = Commits pp xrs
val sign: pp:pparams → cs:eq (list elt){Signed pp cs} → eq dsig
val verify_meter_signature: pp:pparams
  → cs:eq (list elt) → eq dsig → b:eq bool{ b=true ⟹ Signed pp cs }
```

The 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 fsts that takes a list of pairs and returns the list of their first projections.

Given authentic readings xs, the meter function below calls commits, a recursive function that maps sample and commit (specified above) to every element of xs and returns both a list of pairs of readings and openings 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.

```
val meter: pp:pparams → xs:list int{ Readings xs } →
  xrs:(list (x:int ∗ opening pp x)) { xs = fsts xrs } ∗
  cs:eq list elt ∗ eq dsig{ Commits pp xrs = cs }
let meter pp xs = let xrs,cs = commits pp xs in (xrs, cs, sign pp cs)
```

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 make_payment to compute two scalar products: the fee $\sum_t x_t p_t$, and a fee opening $\sum_t o_t p_t$, and pass them to the utility.

```
val make_payment: pp:pparams
  → xrs: (list (x:text ∗ opening pp x)) { Readings (fsts xrs) }
  → ps:eq (list text){| Rates (L ps) ∧ Rates (R ps) ∧
                  SP (fsts (L xrs)) (L ps) = SP (fsts (R xrs)) (R ps) |}
  → (eq text ∗ eq opng)
let make_payment pp xrs ps = let x,r = sums xrs ps in (x,r)
```

The relational pre-condition on the 4th line (SP...) is a *declassification condition*, capturing the user's intent to publish the fee, computed as the scalar product 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 sums, we get the same equation for the openings, showing that the fee opening is then also public. The result type of 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 $1/2$); 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 $1/2$. 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_t; uses the rates p_t to compute the product of exponentials

$$\prod_t {c_t}^{p_t} = \prod_t (g^{x_t} h^{o_t})^{p_t} = \prod_t g^{x_t p_t} h^{o_t p_t} = g^{\sum_t x_t p_t} h^{\sum_t o_t p_t}$$

and compares it to the commitment $g^x h^o$ 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.

```
val verify_payment: pp:pparams
  → ps:eq (list int){Rates ps}
  → cs:eq (list elt) → s:eq dsig (∗ from the meter ∗)
  → x:eq text → r:eq opng (∗ from the user ∗)
  → b:eq bool{ b=true ⟹ ∃xs. Readings xs ∧ x=SP xs ps }
let verify_payment pp ps cs s x r =
  verify_meter_signature pp cs s ∧
  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 RF* is based on the monadic representation of probabilities used in [5, 36]. Our semantics is confined to discrete subdistributions; 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 [14, 34]. An alternative approach is to embed probabilistic programming in a general purpose language, as done e.g. in [28].

Reif [37], Kozen [29], and Feldman and Harel [22] were among the first to develop logics for reasoning about probabilistic programs. Similar logics were later developed by McIver and Morgan [32], and more recently by Chadha et al. [16]. Hurd [26] provides a formalization of the framework of McIver and Morgan [32] 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 [10]. It was later extended to probabilistic procedural programs with adversarial code, and used to formally verify reductionist security proofs of cryptographic constructions [6] and differential privacy of randomized algorithms [9]. 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 [41]; 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 in principle it could be extended to programs with higher-order store using an axiomatic extension of Coq [42]. In contrast, we formalize in Coq a core fragment of RF*, and rest on the F* infrastructure to verify large programs. Our formalization is restricted to programs with first-order store; as we adopt a deep embedding, our formalization could in principle be extended to higher-order store using recent developments in step-indexed semantics [3]. Beyond RHTT and HTT, there have been many efforts to develop and sometimes machine-check program logics for higher-order stateful programs; see [38] for an account of the field.

Relational logics can also be used to reason about continuity [17]. Naturally, numerous program analyses and specialized relational logics enforce 2-properties of programs.

PROTOCOL VERIFICATION. Blanchet's recent account of the field of protocol verification provides a panorama of existing tools and of major achievements [13]. Most of the literature focuses on verifying protocol specifications, or protocol implementations through model extractors [2]; alternatives include generating implementations from verified models [15]. Our work is most closely related to approaches that reason directly about implementations in the symbolic [11, 21] or computational models [12, 23, 30].

CONCLUSIONS. RF* is a full-fledged programming language that supports fine-grained relational reasoning about probabilistic programs, and mechanisms to exploit localized guarantees obtained by such means in global program analyses. As future work, we aim to apply RF* to certify the end-to-end security of large protocol implementations, such as TLS [12], and to verify cryptographic implementations and advanced cryptographic constructions that are inherently higher-order, and hence out of reach for prior relational tools. Examples of such "higher-order cryptography" include leakage-resilience, which accounts for side channel attacks, and key-dependent message security.

References

[1] M. Abadi and C. Fournet. Private authentication. *Theor. Comput. Sci.*, 322(3):427–476, 2004.

[2] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of C protocol implementations by symbolic execution. In *CCS 2012*, pages 712–723. ACM, 2012.

[3] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[4] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied Pi calculus. In *CSF 2010*, pages 107–121. IEEE Computer Society, 2010.

[5] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.

[6] G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *POPL 2009*, pages 90–101. ACM, 2009.

[7] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[8] G. Barthe, B. Grégoire, Y. Lakhnech, and S. Zanella-Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology - CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[9] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL 2012*, pages 97–110. ACM, 2012.

[10] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*, pages 14–25. ACM, 2004.

[11] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL 2010*, pages 445–456. ACM, 2010.

[12] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, S&P 2013*, pages 445–459. IEEE Computer Society, 2013.

[13] B. Blanchet. Security protocol verification: Symbolic and computational models. In *POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2012.

[14] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for bayesian machine learning. In *ESOP 2011*, volume 6602 of *Lecture Notes in Computer Science*, pages 77–96. Springer, 2011.

[15] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *ARES 2012*, pages 65–74. IEEE Computer Society, 2012.

[16] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. Reasoning about probabilistic sequential programs. *Theor. Comput. Sci.*, 379(1-2):142–165, 2007.

[17] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012.

[18] T. Chothia and V. Smirnov. A traceability attack against e-passports. In *FC 2010*, volume 6052 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2010.

[19] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. of Comput. Sec.*, 18(6):1157–1210, 2010.

[20] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[21] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *CSF 2011*, pages 3–17. IEEE Computer Society, 2011.

[22] Y. A. Feldman and D. Harel. A probabilistic dynamic logic. *J. Comput. Syst. Sci.*, 28(2):193–215, 1984.

[23] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *CCS 2011*, pages 341–350. ACM, 2011.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982*, pages 11–20. IEEE Computer Society, 1982.

[25] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical Report RR-6455, INRIA, 2008.

[26] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.*, 346(1):96–112, 2005.

[27] B. Jonsson, W. Yi, and K. G. Larsen. Probabilistic extensions of process algebras. In *Handbook of Process Algebra*, pages 685–710. Elsevier, 2001.

[28] O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2009.

[29] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[30] R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of Java-like programs. In *CSF 2012*, pages 198–212. IEEE Computer Society, 2012.

[31] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[32] A. McIver and C. Morgan. *Abstraction, Refinement, and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[33] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP 2008*, pages 229–240. ACM, 2008.

[34] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL 2005*, pages 171–182. ACM, 2005.

[35] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology - CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

[36] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL 2002*, pages 154–165. ACM, 2002.

[37] J. H. Reif. Logics for probabilistic programming (extended abstract). In *STOC 1980*, pages 8–13. ACM, 1980.

[38] B. Reus and N. Charlton. A guide to program logics for higher-order store, 2012. Unpublished manuscript.

[39] A. Rial and G. Danezis. Privacy-preserving smart metering. In *WPES 2011*, pages 49–60. ACM, 2011.

[40] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1): 5–19, 2003.

[41] G. Stewart, A. Banerjee, and A. Nanevski. Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures. In *PPDP 2013*. ACM, 2013. To appear.

[42] K. Svendsen, L. Birkedal, and A. Nanevski. Partiality, state and dependent types. In *TLCA 2011*, pages 198–212. Springer, 2011.

[43] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP 2011*, pages 266–278. ACM, 2011.

[44] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *PLDI 2013*, pages 387–398. ACM, 2013.