Probabilistic Relational Verification for Cryptographic Implementations

Draft, January 18, 2013.

Gilles Barthe
IMDEA

Cédric Fournet
Microsoft Research

Benjamin Grégoire
INRIA

Pierre-Yves Strub
MSR-INRIA & IMDEA

Nikhil Swamy
Microsoft Research

Santiago Zanella Beguelin
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 new 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 new probabilistic, relational Hoare logic, formalized in Coq—the first such logic for a higher-order, stateful language. We prove the soundness of this logic against a new 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, for example, its abstraction facilities that support 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, noninterference [17], 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-based properties relate a program to an imperfect simulation of it, characterizing the probability of failure, while the security of many cryptographic constructions is specified as a bound on the winning probability of an adversary in a probabilistic experiment.

Recognizing the importance to computer security of such *hyperproperties* [13], 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 [5] is a tool that implements a relational Hoare logic [7] over probabilistic imperative programs. The logic is able to justify common patterns of probabilistic reasoning about hyperproperties used in cryptographic proofs, including observational equivalence arguments, equivalence up to failure events, and reduction steps. As such, EasyCrypt provides a machine-checked framework for proving the security of cryptographic constructions in the computational model, and, over the last two years, has been used for verifying public-key encryption and signature schemes, modes of operation 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 about cryptography (such as the existence of one-way functions) and whose verification encompasses all aspects of the system implementation. Still, the formalisms and tools in the state of the art 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, EasyCrypt does not provide modular reasoning principles that allow probabilistic guarantees to be lifted to more abstract specifications that can be reused elsewhere in a program analysis.

On the other hand, another strand of research into code-based security has emerged. Several research groups have put non-relational, program verification systems for general purpose languages (ranging from C to ML) to use in proving properties of cryptographic implementations [8, 15, 16, 29]. Based on security assumptions on cryptographic libraries, these systems have been used to carry out modular proofs (to varying extents) on implementations at a relatively large scale. For example, over the last two years, F* [29], a dialect of ML, has been used for verifying over 50,000 lines of code, including implementations of multi-party sessions, web-browser extensions, zero-knowledge protocols, a full-fledged implementation of the Transport Layer Security 1.2 (TLS) standard, 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 complet-

ing the certification of security proofs of critical pieces of Internet infrastructure, such as the implementation of TLS (of which large parts are already verified in F*) using RF*. Technically, this paper makes three broad contributions, discussed next.

1. *A probabilistic relational logic for higher-order stateful 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 probabilistic, relational interpretation. This type system forms a logic for λ_p, the first such logic for higher-order, stateful 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 [14] 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 noninterference to various indistinguishability-based properties for encryption, besides others. Several of our examples make essential use of both 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, a compiler download for RF*, 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 x:t{ϕ} where ϕ is a logical formula. For instance, the code fragment below defines a refined type for non-negative integers, then for integers modulo a prime 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 p = 97 $\Longrightarrow$ 73 < p, which is easily discharged by Z3. Type safety means that, whenever an expression e with type x:t{ϕ} reduces to a value v, then v satisfies the formula $\phi[v/\mathsf{x}]$. The type system also 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, x:t → t', where the formal parameter x:t is in scope in the codomain t'. We also use dependent pairs (x:t ∗ t'), where the name x of the first component is in scope in the type t' of the second. For instance, we may write 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 [28], where one can reason about programs using variants of the Hoare state monad of Nanevski et al. [22] together with McCarthy's select/update theory for modeling the heap [21]. 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.

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

We start with a few simple examples. Take the expression e to be z − z; we can give e the type x:int{|L x = R x|}, meaning that for any pair of substitutions σ_L and σ_R, evaluating $\sigma_L e$ yields the same result as evaluating $\sigma_R e$. Relational refinements can also be used to describe properties beyond equivalence, as illustrated by the type x:int → y:int {| L x ≤ R x $\Longrightarrow$ L y ≤ R y |}, of monotonic functions. To describe a k-sensitive integer function with respect to metric dist, we write x:int → y:int {| dist (L y) (R y) ≤ k ∗ dist (L x) (R x) |}. RF* can automatically check (by subtyping) that a function like fun x → k ∗ x is both monotone and k-sensitive for $k \geq 0$.

Relational refinements are strictly more expressive than plain refinements: one can encode any plain refinement { ϕ } as the relational refinement {| L ϕ ∧ R ϕ |} that independently specifies *left* and *right* properties. For instance, the type nat is automatically desugared to n:int {| 0 ≤ L n ∧ 0 ≤ 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 non-classical 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 used to provide a semantic characterization of noninterference-based termination-insensitive information flow controls [27]. Whereas standard type-based information flow controls resort to ad hoc syntactic mechanisms to conservatively determine when a program's observable outputs may depend on its secret inputs, RF⋆ can directly verify the target equivalences. This section provides several examples.

If an expression e that computes over some secret information can be given the type eq a = x:a{|L x = R x|}, then its result can be released safely to an information flow adversary, since the executions of e reveal no information about the secrets. Capturing the intuition from label-based information flow type systems, the type eq a (the type of values that are "equal on both sides") is the type of low-confidentiality values, so we also call it low a. In contrast, hi a is just an alias for a—their values may differ on either side.

Using these types, we can write programs like **fun** x → x − x and give them information flow types like hi int → low int. More interestingly, we can combine such types with logical properties to capture more general policies. For example, a plausible confidentiality policy for credit card numbers conceals all but their last four digits, as 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 dependences (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'. In order to give this program the 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 goals are (1) e:low a, under the assumption that L b = R b = true; (2) e':low a, under the assumption that L b = R b = false; (3) $e \sim e'$:low a, assuming L b = true and R b = false; and (4) $e' \sim e$: low a, assuming L b = false and R b = true.

Our proof rules for relating two values v_0 and v_1 are relatively simple. Proving $v_0 \sim v_1$: x:t{| ϕ |} involves first proving $v_0 \sim v_1$: t (which for base types involves simply showing that both v and v' have type t), and then proving $\phi[v_0/\mathsf{L}\,x][v_1/\mathsf{R}\,x]$. So, RF⋆ can easily prove (**fun** b → **if** b **then** 0 **else** 0) : hi bool → low int, or (**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 harder. 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 write the following program:

```
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). 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 the boolean argument, the resulting heaps are also the same, i.e., the type reveals that f does not leak information despite the side-effect control dependent on the 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 noninterference properties of higher-order stateful programs.

2.3 Sampling, chosen-plaintext security, and one-time pads

We illustrate probabilistic reasoning using symmetric encryption schemes. Our goal is to communicate plaintexts between a sender and a receiver without leaking any information about their content. For simplicity, we assume messages with a fixed size n (called blocks, or fbytes n) and do not involve active attackers. Padding and authentication can be easily added, but would complicate our presentation. We assume that the sender and the receiver share a secret key k (also a block), sampled uniformly at random by calling 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 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
```

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
```

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, from our formal development, we show that it is permissible to give

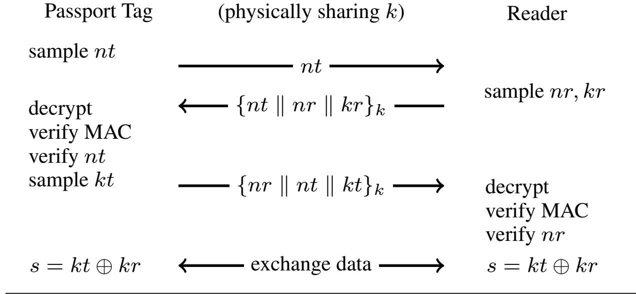

Figure 1. Basic Access Control protocol

sample n a more specific relational type that allows us to complete the proof. In particular, 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 ciphertext is independent of the plaintext. This property is not specific to XOR; indeed, the only relational property that we require of two calls to sample n is that it return a pair of values related by a one-to-one function. As such, relational refinements in RF* capture equivalences on the *distribution of values* computed by a probabilistic program, rather that on the specific values themselves.

To this end, our library provides a polymorphic, typed variant of sample, that takes as additional ghost parameter F, a binary predicate on sampled values (of kind A ⇒ A ⇒ E), whose refinement states that it must be an injective function (or, equivalently, a bijection). Its type declaration in the RF* standard library is

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

In our 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 injective. Section 5 describes more realistic CPA schemes based on sample.

2.4 Implicit flows and passport linkability

Before justifying our typing rules, notably for sample, we present a concrete linkability attack against RFID-equipped passports, recently uncovered by Chothia and Smirnov [12]. 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. The tag has a fixed key k; the reader obtains k by scanning the passport—this step requires physical access to the passport, as it is presented to the reader.

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 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.

French passports reliably return different error messages, whereas other European passports return the same error message, but with measurably different timings. We interpret the attack as an implicit flow of information from the key used to decrypt to the error message. Indeed, if we type our keys as high confidentiality and the nonces and ciphers with eq refinements (inasmuch as 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. However, 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.

```
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 which places various constraints on the structure of the nonce table nts and other mutable locations used in the implementation of this program. Arapinis et al. [2] also analyze this 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 references, probabilistic assignments, and unbounded recursion. The formalization is based on the SSREFLECT extension [18], and on the ALEA library for distributions [3]. Overall, the formalization comprises over 5,000 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 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:

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

where x ranges over a set var of variables, r ranges over a set ref of references and o ranges over a set 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 ($\mathbf{unit}$) along with the types of booleans ($\mathbf{bool}$) and integers ($\mathbf{int}$). 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 $r : \mathsf{ref}\ \mathbf{B}$ 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.

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.

3.2.1 Relational refinement types

Relational assertions are formulae 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 formulae 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. The formal definition of well-formed assertion is as expected, and omitted; we write $G \vdash \Phi : \mathsf{Prop}$ if Φ is a well-formed assertion under G. 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 s.t. $x : \mathbf{int} \in G$, assuming that $r : \mathsf{ref}\ \mathbf{int}$.

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 $x : \mathcal{T}$ s.t. a variable is not bound twice. The refinement type $\mathcal{C}$ is a valid 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 lifted to relational contexts: $\mathcal{G} \lhd G$ is the smallest relation s.t. $[] \lhd []$ and if $\mathcal{G} \lhd G$ and $G \vdash \mathcal{T} \lhd T$ then $\mathcal{G}, x : \mathcal{T} \lhd G, x : T$.

3.2.2 Relational typing

Figure 2 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. 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 initial memories m_1, m_2 satisfying the pre-condition Φ, the pair of distributions over memories obtained by executing e_1 and e_2 are related by the post-condition Ψ (more exactly, by the lifting of Ψ to distributions, defined below). Intuitively, when Ψ is an equivalence relation, 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], which permits to relate an if-expression to an arbitrary expression, is an example of a single-sided rule.

Rules for reference assignment ([REF] and [REF-LEFT]), which come in two flavors too, make use of the ability to write assertions about the resulting memory. For example, the two expressions

$$[\text{Constr}]\ \frac{G \vdash_{\mathsf{v}} b_0 : \mathbf{B} \qquad G \vdash_{\mathsf{v}} b_1 : \mathbf{B} \qquad \mathcal{G} \lhd G}{\mathcal{G} \vdash b_0 \sim b_1 : \mathbf{B}}$$

$$[\text{Var}]\ \frac{\mathcal{G} \lhd G \qquad x \in \text{dom}(\mathcal{G})}{\mathcal{G} \vdash x \sim x : \mathcal{G}(x)}$$

$$[\text{Fun}]\ \frac{\mathcal{G}, x : \mathbf{B} \vdash e_0 \sim e_1 : \mathcal{C} \qquad G \vdash \mathbf{B} \lhd \mathbf{T} \qquad \mathcal{G} \lhd G}{\mathcal{G} \vdash \mathsf{fun}\ x : \mathbf{B} \to e_0 \sim \mathsf{fun}\ x : \mathbf{B} \to e_1 : (x : \mathbf{B}) \to \mathcal{C}}$$

$$[\text{Base-Value}]\ \frac{\mathcal{G} \vdash b_0 \sim b_1 : \mathbf{B}}{\mathcal{G} \vdash b_0 \sim b_1 : \{\Phi[x := b_0, b_1]\} x : \mathcal{U}\{\Phi\}}$$

$$[\text{Ref}]\ \frac{r : \mathsf{ref}\,\mathbf{B} \in \mathcal{G} \qquad \mathcal{G} \vdash b_0 \sim b_1 : \mathbf{B}}{\mathcal{G} \vdash r := b_0 \sim r := b_1 : \{\Phi[x := b_0, b_1]\}\mathbf{unit}\{\Phi[x := r]\}}$$

$$[\text{Ref-Left}]\ \frac{\begin{array}{c} r : \mathsf{ref}\,\mathbf{B} \in \mathcal{G} \qquad \mathcal{G} \lhd G \qquad G \vdash b : \mathbf{B} \\ \mathcal{G} \vdash e_0 \sim e_1 : \mathcal{C}[x_{\mathcal{L}} := b_0] \end{array}}{\mathcal{G} \vdash r := b_0 ; e_0 \sim e_1 : \mathcal{C}[x_{\mathcal{L}} := r_{\mathcal{L}}]}$$

$$[\text{App-Base}]\ \frac{\mathcal{G} \vdash e_0 \sim e_1 : (x : \mathbf{B}) \to \mathcal{C} \qquad \mathcal{G} \vdash b_0 \sim b_1 : \mathbf{B}}{\mathcal{G} \vdash e_0\ b_0 \sim e_1\ b_1 : \mathcal{C}[x := b_0, b_1]}$$

$$[\text{Let}]\ \frac{\begin{array}{c} \mathcal{G} \vdash e_0 \sim e_1 : \{\Phi\}\mathcal{T}\{\Xi\} \\ \mathcal{G}, x : \mathcal{T} \vdash e'_0 \sim e'_1 : \{\Xi\}\mathcal{U}\{\Psi\} \qquad x \notin \text{FV}(\mathcal{U}, \Psi) \end{array}}{\mathcal{G} \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e'_0 \sim \mathsf{let}\ x = e_1\ \mathsf{in}\ e'_1 : \{\Phi\}\mathcal{U}\{\Psi\}}$$

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

$$[\text{If}]\ \frac{\begin{array}{c} \mathcal{G} \lhd G \qquad G \vdash_{\mathsf{v}} v : \mathbf{bool} \\ \mathcal{G} \vdash e_1 \sim e : \{\Phi_1\}\mathcal{C}\{\Psi\} \qquad \mathcal{G} \vdash e_2 \sim e : \{\Phi_2\}\mathcal{C}\{\Psi\} \end{array}}{\mathcal{G} \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \sim e : \{v \Rightarrow \Phi_1 \wedge \neg v \Rightarrow \Phi_2\}\mathcal{C}\{\Psi\}}$$

$$[\text{Flip}]\ \frac{f = x \mapsto x \text{ or } f = x \mapsto \neg x}{\mathcal{G} \vdash \mathsf{flip} \sim \mathsf{flip} : \{\forall y : \mathbf{bool}.\, \Phi[x := y, f(y)]\} x : \mathbf{bool}\{\Phi\}}$$

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

Figure 2. Relational typing rules

$r := b_0$ and $r := b_1$ are related by a post-condition Ψ when Ψ, where all occurrences of $r_{\mathcal{L}}$ (resp. $r_{\mathcal{R}}$) have been replaced by b_0 (resp. b_1), holds as a pre-condition.

Up to now, we only considered rules for expressions headed by non-probabilistic constructions. Rules for random sampling 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

3.3.1 Background

The denotational semantics of well-typed expressions is based on the probability monad [3, 25]. 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 of a monad, by taking as unit and composition operators:

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

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

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

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

Figure 3. Interpretation of values and expressions

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 [19], and used in EasyCrypt to interpret relational judgments. Formally, let $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$; then $P^\sharp\ \mu_1\ \mu_2$ iff:

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

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

3.3.2 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. Then we extend the interpretation to functional types by setting

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

where $\mathsf{M}(X) \triangleq \mathcal{M} \to \mathcal{D}(\mathcal{M} \times X)$.

A valuation I is a function that maps every declaration $x : \mathbf{T}$ to a semantical value. A valuation I is valid 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 $(\!|v|\!)_I$ of a value v and $(\!|e|\!)^m_I$ an expression e are defined in Figure 3. If I is a valid valuation for G, and $G \vdash_{\mathsf{v}} v : \mathbf{T}$ is derivable, then $(\!|v|\!)_I \in [\![\mathbf{T}]\!]$. Likewise, if $G \vdash_{\mathsf{e}} e : \mathbf{T}$ is derivable then $\lambda m.\ (\!|v|\!)^m_I \in \mathsf{M}([\![\mathbf{T}]\!])$.

3.3.3 Relational interpretation

A *relational valuation* $\mathcal{I}$ for G, written $\mathcal{I} \vDash G$, is a pair of 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 $\llparenthesis \Phi \rrparenthesis_{\mathcal{I}}$, s.t. for any formula Φ well-formed under G, for any relation valuation $\mathcal{I} \models F$, $\llparenthesis \Phi \rrparenthesis_{\mathcal{I}}$ 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 4 defines the interpretation of valid relational types, written $\llparenthesis G \vdash \mathcal{T} \lhd \mathbf{T} \rrparenthesis_{\mathcal{I}}$, and valid computation types, written $\llparenthesis G \vdash \mathcal{C} \lhd \mathbf{T} \rrparenthesis_{\mathcal{I}}$, w.r.t. a relational valuation $\mathcal{I}$. A relational valuation $\mathcal{I}$ is valid w.r.t $\mathcal{G}$, written $\mathcal{I} \models \mathcal{G}$, if $\mathcal{G} \lhd G$, and for every variable x declared in G, $\mathcal{I}(x) \in \llparenthesis G \vdash \mathcal{G}(x) \lhd G(x) \rrparenthesis_{\mathcal{I}}$.

Finally, we define the semantic validity of judgments: we say that two values u_1 and u_2 are semantically related in $\mathcal{T}$ under $\mathcal{G}$, written $\mathcal{G} \models v_1 \sim v_2 : \mathcal{T}$, if $\mathcal{G} \lhd G$, and $G \vdash \mathcal{T} \lhd T$, and

$$\forall \mathcal{I} \models \mathcal{G},\ (\llparenthesis u_1 \rrparenthesis_{\pi_1(\mathcal{I})}, \llparenthesis u_2 \rrparenthesis_{\pi_2(\mathcal{I})}) \in \llparenthesis G \vdash \mathcal{T} \lhd \mathbf{T} \rrparenthesis_{\mathcal{I}}$$

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

$$\forall \mathcal{I} \models \mathcal{G},\ (\lambda m.\ \llparenthesis e_1 \rrparenthesis^m_{\pi_1(\mathcal{I})}, \lambda m.\ \llparenthesis e_2 \rrparenthesis^m_{\pi_2(\mathcal{I})}) \in \llparenthesis G \vdash \mathcal{C} \lhd \mathbf{T} \rrparenthesis_{\mathcal{I}}$$

The following theorem states that all judgments of the logic are sound w.r.t. their interpretation.

Theorem 1 (Soundness)**.**

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

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

4. Encoding λ_p in RF*

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 $[\![\Phi]\!]$ $[\![\mathcal{U}]\!]$ $(\lambda_\ _.[\![\Psi]\!])$.[1]

[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.

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 [28]. 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 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. 2 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 rules of Fig. 2 degenerate into the standard typing rules for

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

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

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

Figure 4. Interpretation of relational refinement types

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] of Fig. 2 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] (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 Provings VCs using Z3.

Once a VC has been computed, we ride on an existing encoding of VCs computed for the classic Dijsktra monad within Z3. We rely on a theorem from Schlesinger and Swamy [28] 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 number of lines of code and type annotations (excluding comments), and the typechecking time, which is mostly dominated by the time spent solving VCs in Z3. For lack of space, most of these examples are only briefly described, with a more detailed discussion of the last two programs, counter (a cryptographic construction) and meter (a privacy protocol).

Information flow and passport The first five programs provide many information flow examples and test cases for single-sided rules using several variations of the RDST monad construction of §4. The sixth program, passport, was discussed in §2; its verification involves modelling key-hiding symmetric encryption.

Up to bad The program uptobad illustrates a common cryptographic proof 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. 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 only enforced as long as a distinguished location, bad: **ref** bool, is false. Intuitively, this adds an implicit $\vee$ Bad to every refinement. Independently, we can compute (or bound) the probability of bad being set to true; for passports for instance, we set bad to true when we detect a collision between two sampled nonces nt, and bound its probability with $q^2/2^{64}$ where q is the number of sessions.

Random Oracle The program ro provides an idealized implementation of a cryptographic hash function in the random oracle model. The implementation uses a mutable reference holding a table mapping hash queries made by both honest participants H and adversaries A. To verify the program, we carry several invariants on this table, including proving 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 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 We program an ideal, stateful functionality for CCA2 public-key encryption that maintains a log of prior oracle encryptions, similar to those of Fournet et al. [16]. They require that all code that operates on secrets be placed in a separate module that exports plaintexts as an abstract type. Using relational types, we lift this restriction, enabling us to verify code that use encryption without restructuring. The code is essentially higher-order, simulating ML functors using a dependently typed record of functions.

Nonces and Private Authentication Exploiting the modularity of our CCA2 implementation, we verify simplenonce, a protocol that implements a common authenticated pattern based on fresh nonces, formalizing the intuition "if a encrypts a fresh random 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". We further extend simplenonce to privateauth, a private authentication protocol that allows two parties to communicate securely, protecting both their messages and identities from third parties.

El Gamal encryption The chosen-plaintext security of ElGamal encryption is a classic example of a code-based cryptographic proof. We verify it in RF*, building on an axiomatic theory of cyclic groups.

Name	LOC	TC(s)	Description
arith	43	4.5	Information flow with arithmetic
pure1	35	1.7	Information flow with inference
pure2	33	1.7	Variation of pure1
st	52	3.6	Information flow with state
singlesided	111	14.2	Inf. flow with state and single-sided rules
passport	97	44.2	Defense against attack on French passports
uptobad	15	1.4	Up-to-failure reasoning
ro	73	21.2	Random oracle
cca2	88	6.5	Idealized CCA2 encryption
simplenonce	108	42.5	Authentication with nonces
privateauth	175	81.4	Nonces with private authentication
elg	217	124.5	ElGamal encryption
counter	106	24.1	Counter mode for AES
meter	182	79.8	Pedersen commitments for smart meters
Total	1,378	451.3	

Table 1. Summary of experiments

5.1 Pseudo-random functions and counter-mode

Resuming from the one-time-pad (§2), we implement a more useful symmetric encryption scheme based on a block cipher, such as 3DES and 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 $p_i \oplus m_i$, 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 $p \oplus m_i$ and $p' \oplus m_i$, one trivially obtains $p \oplus p'$, 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
```

Random Initialization Vectors 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) 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 follows:

```
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 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 block cipher is secure with overwhelming probability. 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 that block. Of course, this fails if the mask has already been sampled, so we type f for encryption with a precondition that depends on the current log and requires that i does not occur in the log yet. (We use 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 → sample p
```

When using a 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 fresh random blocks (function encrypt'), the situation is more complex, as there is a non-null probability that two different encryptions pick 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 our scheme with a loss of $q^2/2^{128}$ in the reduction to PRF.

5.2 Privacy-preserving smart metering & billing

We have implemented and verified the "fast billing" protocol of Rial and Danezis [26], which involves recursive data structures and homomorphic commitments. The protocol has three roles: a certified meter that issues signed, fine-grained electricity readings (say one reading every 10 minutes); a utility company that issues 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 his electricity bill. The goal of the protocol is to guarantee the integrity of the monthly fee paid to the utility company and the privacy of the detailed readings. The protocol relies on Pedersen commitments [24] and public-key signatures. 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 implement (and type) commitments, parameterized by some multiplicative group of prime order p. We outline their interface and review their main security properties.

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

val sample: pp:pp → x:text → opening pp x
val commit: pp:pp → 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)
```

Public parameters pp consist of the prime p and two different group generators g and h (possibly chosen by the utility). 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 $\alpha =$ trap pp for specification purposes, in refinement formulas but not in the protocol code. We use α in particular to provide an injective function for randomly sampling the opening o (modulo p) so that it perfectly hides x: the relational refinement type opening in the postcondition of sample records that α∗(L x) + (L o) = α∗(R x) + (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.

Commitments can be multiplied: $g^x h^o * g^{x'} h^{o'} = g^{x+x'} h^{o+o'}$ and exponentiated: $(g^x h^o)^a = g^{ax} h^{ao}$ to compute linear combinations of their exponents without necessarily knowing them. These operations preserve eq and opening relational refinements.

Smart Meter We show some typed code for each role of the protocol. We have abstract predicates Readings and Rates to specify authentic lists of readings and rates. We rely on a signature scheme, assumed resistant against existential forgery attacks. As explained in [16], we express this property using (non-relational) refinements.

```
type Signed (pp:pp) (cs:list elt) =
    ∃xrs. Readings (fsts xrs) ∧ cs = Commits pp xrs
val sign: pp:pp → cs:eq (list elt){Signed pp cs} → eq dsig
val verify_meter_signature: pp:pp
  → 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 precondition for signing (at the meter) and a postcondition of signature verification.

Given authentic readings xs, the meter calls commits, a recursive function that maps commit to every element of xs and returns both a private list of pairs 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 does not convey information about the readings.

```
val meter: pp:pp → 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 From the list of pairs x_t, o_t and the list of rates p_t from the utility, the user computes two scalar products $\sum_t x_t p_t$ and $\sum_t o_t p_t$.

```
val make_payment: pp:pp
  → xrs: (list (x:text ∗ opening pp x)) { Readings (fsts xrs) }
  → ps:eq (list text){ Rates 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 part of the pre-condition (enclosed with (|...|)) is a *declassification condition*, capturing the user's intent to leak the total payment, computed as the scalar product (SP) of the detailed readings and rates. By typing the code of the double scalar product sums, we get the same equation for the openings. The result type of make_payment tells us that those two scalars reveal no further information on any readings leading to the same fee.

Utility The utility verifies the signature on the commitments c_t, and uses the rates p_t to computes 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 values x and 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.

6. Related work and conclusions

Our work spans protocol and relational program verification.

Protocol verification Blanchet's recent account of the field of protocol verification provides a panorama of existing tools and of major achievements [9]. Most of the literature focuses on verifying protocol specifications, or protocol implementations through model extractors; however, our work is most closely related to approaches that reason directly about implementations in the symbolic [8, 15] or computational models [16, 20]. Alternatives include generating implementations from verified models [10], and extracting models from implementations [1].

Relational program verification Relational Hoare Logic was first introduced to reason about program optimizations and information flow of core imperative programs [7]. It was later extended to probabilistic programs, and used to reason about cryptography and differential privacy [4, 6]; and to higher-order programs with store (but not probabilities) and used to reason about advanced information flow policies using interactive proofs in Coq [23]. Relational logics can also be used to reason about continuity [11]. Naturally, numerous program analyses and specialized relational logics enforce 2-properties of programs.

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. We aim in the future to apply RF* to certify the end-to-end security of large protocol implementations, such as ongoing efforts with F* to verify TLS 1.2. Along another axis, we aim to use RF* to verify cryptographic implementations and advanced cryptographic constructions that are inherently higher-order and hence out of reach for existing relational tools, like EasyCrypt. Examples of such "higher-order cryptography" include leakage-resilience, which accounts for side channel attacks, and key-dependent message security.

References

[1] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of c protocol implementations by symbolic execution. In *CCS*, 2012.

[2] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *CSF*, 2010.

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

[4] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, 2009.

[5] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, 2011.

[6] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic reasoning for differential privacy. In *POPL*, 2012.

[7] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[8] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, 2010.

[9] B. Blanchet. Security protocol verification: Symbolic and computational models. In *POST*, 2012.

[10] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *AReS*, 2012.

[11] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[12] T. Chothia and V. Smirnov. A traceability attack against e-passports. In *FCDS*, 2010.

[13] M. Clarkson and F. Schneider. Hyperproperties. *JCS*, 18(6), 2010.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[15] F. Dupressoir, A. Gordon, J. Jürjens, and D. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *CSF*, 2011.

[16] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *CCS*, 2011.

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE S&P*, 1982.

[18] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

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

[20] R. Küsters, T. Truderung, and J. Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *CSF*, 2012.

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

[22] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, 2008.

[23] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE S&P*, 2011.

[24] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1992.

[25] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, 2002.

[26] A. Rial and G. Danezis. Privacy-preserving smart metering. In *WPES*, 2011.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

[28] C. Schlesinger and N. Swamy. Verification condition generation using the Dijkstra state monad. Technical report, Microsoft Research, 2011.

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