Leakage-Free Probabilistic Jasmin Programs

José Bacelar Almeida
jba@di.uminho.pt
HASLab — INESC TEC
Portugal
University of Minho
Portugal

Denis Firsov
denis.firsov@taltech.ee
Tallinn University of Technology
Estonia
Input Output
Estonia

Tiago Oliveira
tiago.oliveira@sandboxquantum.com
SandboxAQ
USA

Dominique Unruh
leakage.k31fea@rwth.unruh.de
RWTH Aachen
Germany
University of Tartu
Estonia

Abstract

This paper presents a semantic characterization of leakage-freeness through timing side-channels for Jasmin programs. Our characterization covers probabilistic Jasmin programs that are not constant-time. In addition, we provide a characterization in terms of probabilistic relational Hoare logic and prove the equivalence between both definitions. We also prove that our new characterizations are compositional and relate our new definitions to existing ones from prior work, which could only be applied to deterministic programs.

To provide practical evidence, we use the Jasmin framework to develop a rejection sampling algorithm and provide an EasyCrypt proof that ensures the algorithm's implementation is leakage-free while not being constant-time.

CCS Concepts: • **Security and privacy → Formal security models**; **Logic and verification**.

Keywords: cryptography, formal methods, EasyCrypt, leakage-freeness, side-channels, timing attack, rejection sampling, Jasmin

ACM Reference Format:
José Bacelar Almeida, Denis Firsov, Tiago Oliveira, and Dominique Unruh. 2025. Leakage-Free Probabilistic Jasmin Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3703595.3705871

1 Introduction

Cryptographic proofs are hard. Implementations are buggy.

When developing and deploying cryptographic systems, we are faced with these two challenges. Cryptographic security proofs tend to be hand-written mathematical proofs, likely containing oversights and other mistakes. They will be read by other humans who may also often overlook those mistakes, especially if they are buried in a high level of detail. In addition, even if a cryptographic scheme is indeed secure, its proof correct, and the underlying computational assumptions unbroken, the final implementation may still contain bugs: Translating an abstract specification into actual code is an error-prone process in itself, leading to new bugs in the final code, making the security proof in the abstract cryptographic setting inapplicable. And finally, adding insult to injury, even if we manage to make code that indeed exactly implements what the specification requires, we could face insecurity due to side-channel attacks. E.g., the code may leak information about our secrets because its runtime depends on some bits of the secret.

The EasyCrypt [10] and Jasmin [3] frameworks aim to resolve this issue. EasyCrypt is a tool in which we can write cryptographic security proofs and verify them using the computer, ensuring high-reliability proofs.[1] However, EasyCrypt does not address implementation issues. The schemes are written in a high level language, very different from what we would find in an actual implementation. Jasmin addresses the implementation side. It consists of an assembly-like language and a certified compiler. In Jasmin, we can write a highly optimized implementation of some cryptographic function and have it compiled to assembly with guarantees of semantic preservation. In addition, the Jasmin compiler can produce an EasyCrypt model for the semantics of the source program. This allows one to 1) perform detailed functional correctness proofs of the source program or 2) link cryptographic security proofs developed in EasyCrypt to the actual binary

[1] This is not perfect, of course. There remains the issue that EasyCrypt itself can have soundness bugs. Or that the security properties are formulated incorrectly. Or that we use a broken cryptographic assumption. These problems are beyond the scope of this work.

CPP '25, January 20–21, 2025, Denver, CO, USA

ACM ISBN 979-8-4007-1347-7/25/01
https://doi.org/10.1145/3703595.3705871

implementations (relying on the Jasmin compiler correctness guarantees).

But Jasmin goes further than that: The exported EasyCrypt code might also contain instructions that explicitly describe side-channel leakage that happens when executing the source (e.g., timing leakage). Then, again, in EasyCrypt, we can prove that the leakage does not depend on the secret inputs. Combining it with a leakage preservation result of the compiler infrastructure would allow us to assert that the executable implementation does not leak.

Putting these pieces together, we can get end-to-end verified implementations of cryptographic algorithms, taking into account everything from the security property to implementation bugs and side-channel security.

A recent addition to the Jasmin language was the support for external system calls for randomness generation (the `#randombytes` primitive). Therefore, the execution of Jasmin programs is no longer deterministic, and both the leakage and results might depend on internal random samples performed during execution. One less obvious consequence of the extension is that the security types of the outputs become sensitive when assessing if a program leaks sensitive information. It's possible that the output of a program might be considered "secret" (for example, in a key-generation procedure), and leakage should not provide information about it. The main research question we want to address in this paper is to assess the impact of that change on the workflow described above as well as propose fixes to the definitions and practices to recover the leakage-freedom guarantees for the end programs.

To motivate and test our approach, we consider the rejection sampling algorithm. Rejection sampling is a widely used tool in cryptography, both directly for sampling values according to distributions not immediately available (i.e. on random bytes) or embedded in techniques such as Fiat-Shamir with Aborts, used in various post-quantum signatures schemes such as the newly standardized ML-DSA (Dilithium) [15].

Contributions. Our technical contributions include the following results:

- We give two alternative semantic characterizations of leakage-freeness for probabilistic Jasmin programs (Sec. 3.1).
- We prove equivalence and compositionality of our leakage-freeness characterizations and relate them to the constant-time definition from prior work (Sec. 3.3).
- We implement a generic rejection sampling in EasyCrypt with proofs of its correctness and termination (Sec. 4.1).
- We implement uniform sampling in Jasmin as a special case of rejection sampling (Sec. 4.2).
- We present two alternative derivations of leakages-freeness for Jasmin's uniform sampling (Sec. 4.3 and Sec. 4.4).

Throughout this work, we have striven to make our results general and reproducible. We tried to make sure that the overall structure of our results is clean and simple to understand, and explained them in this paper in a way that makes it easy to understand to enable future work on other algorithms that follows our work.

Our Jasmin and EasyCrypt developments are made available as supplementary material [1, 2].

1.1 Related Work

The Jasmin toolchain was introduced [3] as a language targeted for the production of high-assurance and high-speed cryptographic software. The Jasmin framework has been combined with the EasyCrypt theorem prover in [5] to establish both the functional correctness and leakage-freeness of high-speed cryptographic implementations of the ChaCha20 stream cipher and Poly1309 message authentication code. Several other cryptographic algorithms have been formalized since then with the Jasmin/EasyCrypt framework, such as Keccak/SHA3 [4], the MPC-in-the-head protocol in [6], and Kyber/ML-KEM [7, 8].

In these works, leakage-freeness amounts to enforce what is commonly referred to as the *cryptographic constant-time* policy — it forbids branching and memory accesses that depend on secret values. In other words, it considers that the control flow (i.e., the program counter) and the addresses of memory accesses are leaked. In a deterministic language, it implies that the execution time is indeed constant. Other leakage models have also been considered in [13, 18] (e.g., leaking the cache line or modeling variable time assembler instructions), but we note that the underlying setting was still a deterministic language. In our presentation, we shall stick to the standard constant-time policy (aka "baseline model"), but other leakage models apply as well.

The formalization of ML-KEM of [7, 8] deserves further mention as it includes a rejection sampling procedure similar to what we consider in this paper. However, rejection sampling is used there during the public key's expansion; hence, the possibility of leaking the output to the adversary is not a concern. As such, the problem that we are interested in looking at in this paper is actually avoided, even if, strictly speaking, the execution time is no longer constant.

Constant-time verification has been recently extended to cover some variants of architectural speculative execution attacks [19]. There, a sound type-system has been proposed to ensure protection against spectre-v1 attacks, but the semantics of the underlying language departs from the model extracted to EasyCrypt by the Jasmin compiler, which blurs the interplay with the analysis performed in this work.

Departing from the Jasmin/EasyCrypt ecosystem, a vast amount of work has considered the general problem of qualitative/quantitative information flows in imperative programs (see [17] for a survey). Most of these works have been oriented towards the design of security type-systems or automatic verification tools capable of preventing or quantifying information-flows in different scenarios. The specific case of cryptographic constant-time verification has been an extremely successful domain of application of these approaches (including the Jasmin realm, e.g. [13, 19]). The more foundational methodology offered by the Jasmin/EasyCrypt workflow adopted in this paper can be used to supplement those analyses for specific cases that fall beyond the scope of those tools.

A general information leakage model based on discrete-time Markov chains has been presented in [14]. It supports arbitrarily located leakage and observation points in an imperative probabilistic program. The formulation of leakage-freeness adopted in this paper can be cast as a special case of their approach, with leakage/observation points dictated by the constant-time policy.

2 Preliminaries

2.1 EasyCrypt

EasyCrypt (EC) is an interactive framework for verifying the security of cryptographic protocols in the computational model. In EasyCrypt security goals and cryptographic assumptions are modelled as probabilistic programs (a.k.a. games) with abstract (unspecified) adversarial code. EasyCrypt supports common patterns of reasoning from the game-based approach, which decomposes proofs into a sequence of steps that are usually easier to understand and to check [11].

To our readers who are not familiar with EasyCrypt, we also suggest reading a short introduction to EasyCrypt in [16, Section 2]. More information on EasyCrypt can be found in the EasyCrypt tutorial [11].

To readers who are familiar with EasyCrypt we only give a brief overview of our syntactical conventions: we write $\leftarrow$ for `<-`, $\xleftarrow{\$}$ for `<$`, $\xleftarrow{@}$ for `<@`, $\wedge$ for `/\`, $\vee$ for `\/`, $\leq$ for `<=`, $\geq$ for `>=`, $\forall$ for `forall`, $\exists$ for `exists`, $\mathbf{m}$ for `&m`, $\mathcal{G}_A$ for `glob A`, $\mathcal{G}_A^{\mathbf{m}}$ for `(glob A){m}`, $\lambda x.\, x$ for `fun x => x`, $\times$ for `*`. Furthermore, in `Pr`-expressions, in abuse of notation, we allow sequences of statements instead of a single procedure call. It is to be understood that this is shorthand for defining an auxiliary wrapper procedure containing those statements.

2.2 Jasmin

Jasmin is a toolchain for high-assurance and high-speed cryptography [3, 5]. The ultimate goal for Jasmin implementations is to be efficient, correct, and secure. The Jasmin programming language follows the "assembly in the head" programming paradigm. The programmers have access to low-level details such as instruction selection and scheduling, but also can use higher-level abstractions like variables, functions, arrays, loops, and others.

The semantics of Jasmin programs is formally defined in Coq to allow users to rigorously reason about programs. The Jasmin compiler produces predictable assembler code to ensure that the use of high-level abstractions does not result in run-time penalty. The Jasmin compiler is verified for correctness. This justifies that many properties proved about the source program will carry over to the corresponding assembly (e.g., safety, termination, functional correctness).

The Jasmin workbench uses the EasyCrypt theorem prover for formal verification. Jasmin programs can be extracted to EasyCrypt to address functional correctness, cryptographic security, or security against timing attacks.

2.2.1 Jasmin Basics.

We explain the basics and workflow of Jasmin development in a simple example. More specifically, our goal is to implement a procedure that samples uniformly a secret bit (encoded as a byte `0` or `1`). Below is the "naive" attempt:

```
inline fn random_bit_naive() → reg u8{
   stack u8[1] byte_p;
   reg u8 r;

   byte_p = #randombytes(byte_p);
   if (byte_p[0] < 128){
     r = 0;
   }else{
     r = 1;
   }
   return r;
}
```

The program has no arguments and outputs an unsigned byte in a register (type `reg u8`). The body of the program starts by declaring the variables and their respective types. In particular, we declare a variable `byte_p` of type `stack u8[1]` which has an effect of allocating a memory region on the stack. Next, we generate a random byte with a system call `#randombytes`. The system call takes the byte array as its argument and fills its entries with randomly generated bytes. In this way, we sample a single random byte into local variable `byte_p[0]`. Hence, with probability `1/2` the value `byte_p[0]` is smaller than `128` and the result of computation is `0`; otherwise, we return the value `1`.

To address correctness of `random_bit_naive` we can instruct the Jasmin compiler to extract an EasyCrypt model of `random_bit_naive` program. This produces a module, renamed for convenience as `XtrI`, defining the procedure `random_bit_naive`. Jasmin extracts programs to EasyCrypt by systematically translating all datatypes and Jasmin programming constructs. See the code below.

```
module type Syscall_t = {
```

```
 proc randombytes_1(b:W8.t Array1.t): W8.t Array1.t
}.

module SC_D : Syscall_t = {
 proc randombytes_1(a:W8.t Array1.t)
                      : W8.t Array1.t = {
  a <$- dmap WArray1.darray
   (λ a ⇒ Array1.init (λ i ⇒ WArray1.get8 a i));
  return a;
 }
}.

module XtrI(SC:Syscall_t) = {
 proc random_bit_naive () : W8.t = {
   var r:W8.t;
   var byte_p:W8.t Array1.t;
   byte_p  ← witness;
   byte_p <@- SC.randombytes_1 (byte_p);
   if ((byte_p.[0] < (W8.of_int  128))) {
     r ← (W8.of_int  0);
   } else {
     r ← (W8.of_int  1);
   }
   return (r);
 }
}.
```

For example, Jasmin datatype `reg u8` of 8-bit words was translated to the EasyCrypt type `W8.t`. The type of a single-entry 8-bit array `stack u8[1]` was translated to `W8.t Array1.t`. In the code above, `witness` denotes an "arbitrary" uninterpreted value of type `W8.t Array1.t`.

Since `random_bit_naive` uses the system call `#randombytes`, Jasmin parametrized the extracted module (`XtrI`) by a "system call provider" `SC`, that includes procedures for all the `#randombytes` instances needed by the program (in our example, only the $\texttt{SC.randombytes}_1$). Such an indirection allows users to choose their own interpretation of system calls, other than the "default" interpretation $\texttt{SC}_\texttt{D}$, also generated by the compiler, which models `#randombytes` as a truly random byte generator (e.g. one could interpret `#randombytes` as an invocation of a pseudo-random generator). In our development, we stick to the default interpretation $\texttt{SC}_\texttt{D}$.

The main purpose of the EasyCrypt's module `XtrI` is to address the correctness of the Jasmin's implementation. More specifically, we can use the EasyCrypt's built-in probabilistic Hoare logic to prove that `random_bit_naive` returns values `0` and `1` with probabilities equal to `1/2`.

2.2.2 Leakage-Freeness. Another important aspect of the Jasmin framework is that it allows users to analyze whether the implementation is "leakage-free". Intuitively, the program is "leakage-free" if its execution time does not leak any additional information about its secrets. To perform leakage-free analysis, a user would instruct Jasmin compiler to extract a program to EasyCrypt with leakage annotations. In this case, the resulting EasyCrypt module which we renamed as `XtrR`, has a global variable `leakages`, which is used in the extracted EasyCrypt procedures to accumulate information that gets leaked in case of a timing attack. For example, if we extract `random_bit_naive` to EasyCrypt with leakage annotations, the result is as follows:

```
module XtrR(SC:Syscall_t) = {
 var leakages : leakages_t // global variable
 proc random_bit_naive () : W8.t = {
   var r, aux_0: W8.t;
   var aux, byte_p:W8.t Array1.t;

   byte_p ← witness;
   leakages ← LeakAddr([]) :: leakages;
   aux <@- SC.randombytes_1 (byte_p);
   byte_p ← aux;

   leakages ←
     LeakCond((byte_p.[0] < (W8.of_int  128)))
        :: LeakAddr([0]) :: leakages;

   if ((byte_p.[0] < (W8.of_int  128))) {
     leakages ← LeakAddr([]) :: leakages;
     aux_0 ← (W8.of_int  0);
     r ← aux_0;
   } else {
     leakages ← LeakAddr([]) :: leakages;
     aux_0 ← (W8.of_int  1);
     r ← aux_0;
   }
   return (r);
 }
}.
```

The type `leakages_t` is a list of leaked atoms, and the entries in the `leakages` accumulator must be understood as data which an attacker could learn when carrying out a timing attack. The internal structure of the `leakages` is not really important at this stage, but we note that `LeakAddr([0])` reflects the access to index `0` of the array; `LeakAddr []` the access to the stack variable; and `LeakCond(...)` the result of evaluating the `if`-statement condition.[2] Since the execution path fully determines the (assumed secret) output, we must conclude that the implementation of `random_bit_naive` is not leakage-free.

Let us implement a procedure `random_bit` which gets rid of the problematic `if`-statement:

```
inline fn random_bit() → reg u8{
   stack u8[1] byte_p;
   reg u8 r;
   byte_p = #randombytes(byte_p);
```

[2] In general, conditionals (if-statements) result in executing different blocks of code which typically leads to the significant leakage (timing difference). Therefore, the Boolean value which identifies the activated branch is added to the leakage accumulator.

```
  r = byte_p[0];
  r &= 1;
  return r;
}
```

In `random_bit` definition we convert a random byte `byte_p[0]` to the values `0` or `1` by doing a bitwise "and" operation of `byte_p[0]` with value `1` and return the result. Again, we can prove that the new version of `random_bit` is a uniform distribution of values `0` and `1`. More interestingly, we note that this new version is leakage-free. In fact, after extraction to EasyCrypt with leakage-annotations we get the following EasyCrypt code:

```
module XtrR(SC:Syscall_t) = {
 var leakages : leakages_t

 proc random_bit () : W8.t = {
   var r, aux_0: W8.t;
   var aux, byte_p: W8.t Array1.t;

   byte_p ← witness;
   leakages ← LeakAddr([]) :: leakages;

   aux @← SC.randombytes_1 (byte_p);
   byte_p ← aux;
   leakages ← LeakAddr([0]) :: leakages;
   aux_0 ← byte_p.[0];
   r ← aux_0;
   leakages ← LeakAddr([]) :: leakages;
   aux_0 ← (r `&` (W8.of_int 1));
   r ← aux_0;
   return (r);
 }
}.
```

Now, upon execution of `random_bit`, the `leakages` accumulator does not contain any data specific to the output of the program. Indeed, the leakage generated in any execution of `random_bit` is fixed.

Notice, however, that what concerns us is not the distinction between deterministic vs. probabilistic leakage, but rather if the leakage reveals information about the secrets. To argue formally about it, we must rely on rigorous definitions of leakage-freeness and cryptographic constant-time (see Sec. 3).

3 Leakage-Freeness and Constant-Time

We consider a collection of Jasmin procedures that are extracted into EC in two modes: `XtrI` and `XtrR`. Each one of these is a module that includes the EC's model of Jasmin-implemented functions:

- `XtrI.f` - an EC procedure modeling the input/output behaviour of Jasmin function `f` (hence, calling it an abstract or "Ideal" setting). This is a stateless module (Jasmin does not have global variables).
- `XtrR.f` - an EC procedure that, in addition to the input/output behavior, models also what is leaked during execution (accumulated in variable `XtrR.leakages`). This is what we call the concrete or "Real" setting.

We are interested in programs `f` whose output is deemed secret, with both public and secret inputs (denoted by `pin` and `sin`, respectively). As a meta-property of the extraction mechanism, the marginal probability distribution of the result in `XtrR.f` agree with the probability distribution induced by `XtrI.f`. This property can be stated as an equivalence of programs in the *probabilistic Relational Hoare Logic (*pRHL*)*, namely:

$$\texttt{XtrI.f} \sim \texttt{XtrR.f}: \quad =\{pin, sin\} \Longrightarrow =\{res\}. \tag{1}$$

Here, $=\{res\}$ denotes the equality of outputs of the left (`XtrI.f`) and the right (`XtrR.f`) programs. The proof of this property is not produced automatically but can be easily confirmed in EC for concrete instances as it is typically proved automatically resorting to EC's `sim` tactic. Informally, this equivalence asserts that we obtain equally distributed results when running both programs in initial memories that equate the values of the inputs (`pin` and `sin`).

Before Jasmin was extended with `#randombytes` primitive, all its programs were deterministic. In this case, proving that a program is leakage-free (or "constant-time" in the parlance of the prior work) requires only to prove that the probability of producing a particular leakage does not depend on the secret input. The formal definition is as follows:

Definition 3.1 (Constant-time Deterministic Programs)**.** *Let* `f` *be a total deterministic Jasmin program and* `XtrR.f` *be the result of its extraction to EasyCrypt with leakage annotations. Then,* `f` *is* constant-time *(abbreviated* CTdef(f)*) when,*

```
∀ sin sin' pin l m,
   Pr[ XtrR.f(pin,sin)@m: XtrR.leakages = l ]
   = Pr[ XtrR.f(pin,sin')@m: XtrR.leakages = l ].
```

In the formula above, `m` denotes the initial memory of the program. We have cast the definition as a probabilistic non-interference property — it asks for the distribution of leakage to be independent of secret inputs (independently of `f` being deterministic or not). Of course, when `f` is deterministic, the leakage is just a function of `f`-inputs, which justifies the `constant-time` in the name (public inputs fully determine execution time).

This formulation is also useful to emphasize the fact that naively lifting the definition to a probabilistic setting fails to capture leakage-freeness for randomized programs with secret outputs, as we have observed when discussing the `random_bit_naive` program (see Sec. 2.2.2). In fact, `random_bit_naive` trivially satisfies the Definition 3.1 from above, since there are no secret inputs.

In the next section, we propose new characterizations of the leakage-freeness for probabilistic programs.

3.1 Leakage-Free Programs

We want to guarantee safety against timing attacks. In other words, we want to ensure that programs which satisfy our notion of leakage-freeness must not leak any information about their secret inputs and the result of their computation through timing attacks. Definition 3.1 must then be strengthened to enforce independence between the output and leakage.

Definition 3.2 (Leakage-Free Jasmin Programs)**.** *Let* `f` *be a total Jasmin program with secret output and* `XtrR.f` *be the result of its extraction to EasyCrypt with leakage annotations. Also, let* `pin` *and* `sin` *be public and secret inputs, respectively. Then,* `f` *is* leakage-free *(abbreviated* LFdef*(f)) when,*

```
∀ s, ∃ g, ∀ sin pin a l m, XtrR.leakages{m} = s
 ⇒ let v = Pr[out ← XtrR.f(pin,sin)@m:
                XtrR.leakages = l ++ s ∧ out = a] in
   let w = Pr[out ← XtrR.f(pin,sin)@m: out = a] in
   0 < w ⇒ v/w = g(pin,l).
```

In the definition above `v/w` denotes a conditional probability of producing leakages `l` given that the output is `a`. Intuitively, the program is leakage-free if there exists a function `g` such that the conditional probability `v/w` can be computed only from public inputs and the leakages `l`. That is the "leakage" distribution does not depend on the secret input `sin` and the result `out`.

To make our definition composable, we allow the leakage accumulator to start from arbitrary initial state `s`. At this point, it is important to realize that computations themselves (i.e., function `XtrR.f`) cannot introspect (i.e., analyze) leakages in `XtrR.leakages` – a consequence of Equation 1 which is ensured by the extraction mechanism.

To apply this definition to `random_bit` (see Sec. 2.2.2) we must define a function that computes the conditional probability in order to instantiate the existential quantifier in the definition above. For the `random_bit`, it could be defined as follows:

```
op g l = let random_bit_l
           = [LeakAddr  []; LeakAddr [0];
              LeakAddr []] in
   if l = random_bit_l then 1 else 0.
```

Here, `g` checks if the list of leakages `l` is well-formed (i.e., equals to a constant list denoted by `random_bit_l`) in which case it returns `1`, and `0` otherwise. Using basic EC reasoning, we can prove that the Jasmin program `random_bit` with function `g` as defined above satisfies the definition of being leakage-free according to Definition 3.2.

At the same time, the function `random_bit_naive` does not satisfy Definition 3.2, as the probability of observing some leakage traces clearly depend on the output.

3.2 pRHL characterization

The advantage of Definition 3.2 is that it has a clear and intuitive semantics in terms of conditional probability. At the same time, it could be cumbersome to prove directly that a program satisfies Definition 3.2 because proof requires us to explicitly describe the contents of the leakages (i.e., we must give the existentially quantified function `g`). This is an inconvenience that contrasts with the simplicity and elegance allowed by the standard *constant-time* characterization of leakage-freeness for deterministic programs. More specifically, the prior work in Jasmin addressed leakage-freeness of deterministic programs by "automatically" proving the following pRHL equivalence in EC:

Definition 3.3 (pRHL Constant-time Deterministic Programs)**.** *A deterministic total program* `f` *is said to be* constant-time *(abbreviated* CT*(f)) when the following program equivalence holds:*

$$\mathsf{XtrR.f} \sim \mathsf{XtrR.f} : =\{pin, \mathsf{XtrR.leakages}\} \Longrightarrow =\{\mathsf{XtrR.leakages}\}.$$

The above is trivially equivalent to Definition 3.1 and is indeed the usual formulation of probabilistic non-interference. The appeal of the above formulation is that it is extremely useful in practice, as it is often proved automatically through the EC's `sim` tactic. It also better reveals more explicitly useful properties such as compositionality.

To overcome the pitfalls detailed for Definition 3.1, we refine the above definition by additionally requiring independence of the output and leakage. It can be enforced in pRHL by self-composition [12].

Definition 3.4 (pRHL Leakage-Freeness)**.** *A total program* `f` *is said to be* leakage-free *(abbreviated* LF*(f)) iff the following equivalence of programs holds:* `∀ pin sin sin'`,

$$\{\texttt{r} \overset{@}{\leftarrow} \texttt{XtrR.f(pin,sin);}\} \sim \left\{\begin{array}{l}\texttt{\_} \overset{@}{\leftarrow} \texttt{XtrR.f(pin,sin');}\\ \texttt{r} \overset{@}{\leftarrow} \texttt{XtrI.f(pin,sin);}\end{array}\right\}$$
$$: =\{pin, sin, \mathsf{XtrR.leakages}\} \Longrightarrow =\{r, \mathsf{XtrR.leakages}\}$$

Notice that on the right-hand side we are using both the plain and instrumented semantics of program `f` (respectively `XtrI.f` and `XtrR.f`). This ensures that the global variable accumulating the leakage is only updated once. Intuitively, we can look at this definition as enforcing the equivalence between a "real world" where the evaluation of `f` leaks, with an "ideal world" that computes the result (without leakage), and simulates the leakage by evaluating the instrumented semantics on some arbitrary secret input `sin'`. As we shall see in the next section the pRHL characterization indeed captures the same property as the Definition 3.2.

The main advantage of pRHL characterization is that in EasyCrypt for leakage-free programs with non-probabilistic

runtime the LF formulation has trivial proofs (see details in Sec. 3.4).

3.3 Properties

We collect now main properties relating the various definitions. They have been fully formalized in EC[3].

Proposition 3.1. *For any given total program* f*, the following implications hold:*

1. $\mathsf{LF}(f) \implies \mathsf{CT}(f)$
2. $\mathsf{det}(f) \Rightarrow (\mathsf{LF}(f) \iff \mathsf{CT}(f))$
3. $\mathsf{LF}(f) \iff \mathsf{LFdef}(f)$
4. $\mathsf{CTdef}(f) \iff \mathsf{CT}(f)$

where $\mathsf{det}(f)$ *is an abbreviation for*

$$\exists\, g, \forall\, p\ s,\ \{\texttt{r} \overset{@}{\leftarrow} \texttt{XtrI.f(p,s);}\} : \texttt{true} \implies \texttt{r = g(p,s)}.$$

Here, determinism is established by a functional specification expressed by a (partial) Hoare triple, whose proof is often a byproduct of the correctness proof.

The first two points support the view of $\mathsf{LF}(f)$ as a generalization of $\mathsf{CT}(f)$ for probabilistic programs. To imply $\mathsf{LFdef}(f)$ from $\mathsf{LF}(f)$, we analyze a function defined by the following expression:

$$\frac{\texttt{Pr[out} \leftarrow \texttt{XtrR.f(pin,sin)@m: out=r} \wedge \texttt{XtrR.leakages=l]}}{\texttt{Pr[out} \leftarrow \texttt{XtrR.f(pin,sin)@m: out=r]}}.$$

The proof of the converse implication is more challenging, as it demands a fairly detailed reasoning on the underlying semantics of both `XtrI.f` and `XtrR.f`. To that end, a key role is played by what is called *reflection lemmas* (see [16] for more details), that have been proved for abstract procedures, and which allow us to bridge assertions established at the procedural level to the underlying semantic distributions (shown here the instance for `XtrR.f`):

```
lemma R_opsemE m': ∃ d, ∀ P _pin _sin m,
   G^m_XtrR = G^m'_XtrR ⇒
   Pr[ out ← XtrR.f(_pin, _sin)@m: P(out, G_XtrR) ]
   = μ (d _pin _sin) P.
```

Above $\mathcal{G}^{\mathbf{m}}_{\mathtt{XtrR}}$ denotes an initial memory of module `XtrR`. And μ `d P` denotes the probability that the predicate `P` holds for values distributed according to `d`. The importance of this lemma is that it allows us to exactly capture the probabilistic semantics of `XtrR.f`.

Additionally, it can be shown that the witness function given by $\mathsf{LFdef}(f)$ is a probability mass function of a distribution on leakages `dLeak` (i.e. the summation of the direct image of any subsets of leakage traces lie in the unit interval). Moreover, the associated condition enforces the equality of distributions

$$\mathsf{dR} = \mathsf{dI} \ \text{`}*\text{`}\ \mathsf{dLeak},$$

where `*` denotes the *product distribution*, and `dR` and `dI` are the *probabilistically reflected* distributions related to `XtrR.f` and `XtrI.f`, respectively. Moving back and forth through *reflection* lemmas, we show that the equality of probabilities needed to establish $\mathsf{LF}(f)$ holds.

We conclude this section by presenting a compositionality result. Intuitively, compositionality allows users to "automatically" conclude leakage-freeness for a composite program from leakage-freeness of its components.

Proposition 3.2 (Compositionality)**.** *Let* f *and* g *be total programs such that:*

- f *expects* `pin` *and* `sin1` *as public and secret inputs respectively, and produces an output* `sout1`*;*
- g *expects* `pin` *and* `(sout1,sin2)` *as public and secret inputs respectively, and produces an output* `sout2`*;*
- *both are leakage-free (i.e.* $\mathsf{LF}(f)$ *and* $\mathsf{LF}(g)$ *holds).*

Then, the program

$$\texttt{h(pin,(sin1,sin2))} \doteq \left\{\begin{array}{l}\texttt{sout1} \overset{@}{\leftarrow} \texttt{f(pin,sin1);}\\ \texttt{sout2} \overset{@}{\leftarrow} \texttt{g(pin,(sout1,sin2);}\\ \texttt{return sout2;}\end{array}\right\}$$

is itself leakage-free $\mathsf{LF}(h)$.

Its proof relies on the observation that `XtrI.f` and `XtrI.g`, being stateless, can be repositioned freely on the right-hand side of the equivalence.

We believe that compositionality is an important property to make our novel definitions practically useful for establishing leakage-freeness for large composite programs and protocols.

3.4 Proof Effort: LF vs LFdef

The main advantage of pRHL characterization is that in EC, for programs whose leakage is syntactically independent of all secrets, the derivation of LF property can be established using simple pRHL reasoning. For example, for `random_bit`, and after instantiating our generic development, the EC proof looks as follows:

```
lemma random_bit_LF:
 equiv[RSim(XtrI,XtrR).main ~ SimR(XtrI,XtrR).main
  : ={pin, sin, G_JR} => ={res, G_JR}].
proof. proc. inline*. wp. rnd.
 wp. rnd. wp. skip. progress.
qed.
```

The above statement is an EC formalization of Definition 3.4 and the proof-script performs a symbolic simulation.

In contrast, when applying Definition 3.2 directly, we must provide an explicit function for calculating leakages and carefully analyzing conditional probabilities. This way, the derivation is tightly coupled with the logic of the program, and any change to the program will necessarily break the proof of the leakage-freeness.

[3]Available on file `proof/LeakageFreeness_Analysis.ec` of the development.

For algorithms whose leakage cannot be syntactically decoupled from secrets, the derivation of leakage-freeness becomes challenging for both formulations. We will return to this subject in Sec. 4, where we analyze leakage-freeness for the rejection sampling algorithm following both strategies.

4 Rejection Sampling

In Jasmin we can use `#randombytes` system call to generate bytes uniformly at random. However, this does not immediately give us uniform distributions on sets whose cardinality is not a power of 2. In this section our goal is to describe a verified (correct and leakage-free) Jasmin implementation of uniform sampling of arbitrary size. One solution to this problem is "rejection sampling". In rejection sampling we are drawing random elements from a given distribution d and rejecting those samples that don't satisfy some predefined criteria. If the sampled element was rejected then we sample again until the element is accepted. For example, if d is a uniform distribution from $[0 \ldots 7]$ and we perform rejection sampling from d with criteria that the resulting element must be smaller than 3 then we can prove that this precisely gives a uniform distribution of 0,1, and 2.

The challenging aspect of rejection sampling is that it does not have an apriori termination time which means that we do not know how long it will take to produce an element which satisfies the criteria. However, we can prove that if the source distribution d has elements which satisfy the criteria then the rejection sampling is always terminating (i.e. terminates with probability 1), but the runtime is probabilistic.

The standard library of EasyCrypt has a formalization of rejection sampling properties in theory `Dexpected.ec`. These cover only functional correctness and do not address the leakage-freeness of the algorithm. Also the proof strategies are different. In our work we derive properties of rejection sampling by solving a recurrence equation which gives us a clean and concise proof of correctness.

In the next section, we continue by implementing a "high-level" rejection sampling algorithm in EC and proving its properties. Next we implement a uniform sampling in Jasmin as a special case of rejection sampling. Next, we extract the Jasmin implementation to EasyCrypt and show that it is correct by establishing equivalence with the "high-level" EasyCrypt implementation (i.e., `RS.rsample` function). Finally, we extract the Jasmin sampling algorithm to EasyCrypt with leakage annotations and present two alternative proofs that it is leakage-free (the proof of leakage-freeness makes use of the correctness and termination proofs).

4.1 Rejection Sampling in EasyCrypt

We start by implementing a rejection sampling algorithm in EasyCrypt. Our algorithm is parameterized by a lossless distribution `d` of a parameter type `X` (we say that a distribution is lossless if sampling from the distribution always terminates). We implement a module `RS` with procedure `rsample(P)`, where `P` is a predicate on the elements of the distribution. In this procedure we run a while loop in which we sample an element `x` from `d` on each iteration. The while-loop terminates when the sampled element `x` satisfies the predicate `P`.

```
type X.
op d : X distr.
axiom d_ll : is_lossless d.

module RS = {
 proc rsample(P : X → bool) : X = {
   var b : bool;
   var x : X;
   x ← witness;
   b ← false;

   while(!b){
    x $← d;
    b ← P x;
  }

  return x;
 }

 proc rsample1(P : X → bool) = {
   var x : X;
   x $← d;
   if(! P x){
     x @← rsample(P);
   }
   return x;
 }
}.
```

To help with the derivation of correctness of `rsample` we also implement `rsample1` procedure which is computationally equivalent to `rsample`, but with the explicit unrolling of the first iteration of the while loop.

Let us now address the correctness and termination of the `RS.rsample` procedure. In the first step, we show that `RS.rsample` and `RS.rsample1` are computationally equivalent. This is easily proved by using pRHL and expanding the while loop in `rsample` with the `unroll` tactic.

```
lemma samples_eq m P Q:
 Pr[x ← RS.rsample(P)@m: Q x]
      = Pr[x ← RS.rsample1(P)@m: Q x].
```

In the next step we express the probability of events of `rsample1` in terms of the probability of the same events of `rsample`. To achieve that we use probabilistic Hoare logic (pHL) and split the total probability into cases which correspond to the branches of the `if`-statement in `rsample1`:

```
lemma rsample1_rsample m P Q:
  Pr[x ← RS.rsample1(P)@m: Q x]
```

```
  = μ d !P * Pr[x ← RS.rsample(P)@m: Q x]
                                  + μ d (Q `∧` P).
```

Now, we can combine `samples_eq` and `rsample1_rsample` and arrive at the following recurrence:

```
lemma rsample_rec m P Q:
⇒ Pr[x ← RS.rsample(P)@m: Q x]
    = μ d !P  *  Pr[x ← RS.rsample(P)@m: Q x]
                                  + μ d (Q `∧` P).
```

If the total probability mass of the predicate `P` is not zero then the above recurrence has the following solution:

```
lemma rsample_pmf_gen m P Q: μ d P ≠ 0
⇒ Pr[x ← RS.rsample(P)@m: Q x]
     = μ d (Q ∧ P)  /  (1 - μ d !P).
```

For the special case when `Q` is a subset of `P` and event `P` has non-zero probability then we arrive at the following equation:

```
lemma rsample_pmf m P Q: (∀ x, Q x ⇒ P x)
⇒ μ d P > 0
⇒ Pr[out ← RS.rsample(P)@m: Q out]
    = μ d Q / μ d P.
```

In this case, the right-hand side of the above equation denotes a conditional probability of `Q` given `P`.

As a simple consequence we get that the procedure `RS.rsample(P)` returns an element `x` which satisfies the predicate `P` with probability `1`. This also means that the procedure `rsample` is terminating (or lossless in the parlance of EasyCrypt):

```
lemma rsample_ll m P: μ d P >  0
⇒ Pr[x ← RS.rsample(P)@m: P x] = 1.
```

4.2 Uniform Sampling in Jasmin

Jasmin lacks expressivity to handle implementation of a generic rejection sampling algorithm (which would be parameterized by predicate `P` and distribution `d`; see Sec. 4.1)[4]. As a result, to perform our case study we instantiate rejection sampling for uniform sampling (which is broadly utilized in cryptographic protocols). We implement a Jasmin function which specializes the predicate `P` to λ `x. x < a` (for a parameter `a`) and uses `#randombytes` system call as a distribution `d`. In this way, we implement a uniform sampling from an interval `[0...a-1]` for a given parameter `a`.

Also in Jasmin language it is impossible to express arrays of parametric length. Therefore, in the preamble of all our Jasmin development we define a constant `nlimbs` and then represent the inputs and outputs of our programs by an arrays of size `nlimbs` of 64-bit unsigned binary words.[5]

Now we describe an implementation of a Jasmin program $\texttt{bn\_rsample}_{\texttt{i}}\texttt{(a)}$ (prefix `bn` stands for big-number) whose input `a` is an `nlimb`-array representing a number from the interval $[0 \ldots 2^{64 \cdot \texttt{nlimbs}} - 1]$ which is allocated on stack. The program returns a pair `(i,p)`, where `i` is a counter of while-loop iterations and `p` is a binary array which represents a number sampled uniformly at random from the interval `[0...a-1]`. In our implementation, the counter `i` is a "logical" variable of type `int` (i.e., unbounded integer) which is only needed to facilitate proving in EasyCrypt. We also define function `bn_rsample(a)` which discards the logical counter `i`.

In the implementation below we run a while-loop and at every iteration we use the system call `#randombytes` to sample a random number `p` from the interval $[0 \ldots 2^{64 \cdot \texttt{nlimbs}} - 1]$. Then we subtract `p` from `a` by using a `bn_subc` function.[6] The result of subtraction is stored in the memory of the first argument of `bn_subc`. Therefore, to preserve the initial value of `p`, we first copy it to the variable `q` by using the `bn_copy` call. Importantly, in addition to the result of subtraction the program `bn_subc` also returns the "carry" flag `cf` which is set to `true` if the first argument is smaller than the second. The while loop is iterated until the flag `cf` is set to `true` which would indicate that the sampled number `p` is smaller than `a` as desired:

```
inline fn bn_rsample_i(stack u64[nlimbs] a)
              → (inline int, stack u64[nlimbs]){
  stack u64[nlimbs] q p;
  reg ptr u64[nlimbs] _p;
  reg bool cf;
  inline int i;
  i = 0;
  p = bn_set0(p);
  _, cf, _, _, _, _ = #set0(); // sets cf to 0
  while (!cf) {
    _p = p;
    p = #randombytes(_p);
    q = bn_copy(p);
    cf, q = bn_subc(q,a);
    i = i + 1;
  }
  return i,p;
}

inline fn bn_rsample(stack u64[nlimbs] a)
                        → (stack u64[nlimbs]){
  stack u64[nlimbs] p;
  _, p = bn_rsample_i(a);
  return p;
```

[4] Jasmin does not have any built-in types of distributions and the only way to generate randomness in Jasmin is by using the `randombytes` system call.

[5] In our work we put `nlimbs := 32`, but our development can be recompiled with any value.

[6] The implementation of `bn_subc` is included into the `libjbn` library.

```
}
```

Next, to address correctness we compile Jasmin code to EasyCrypt without leakage-annotations. This produces a module `XtrI` with the EasyCrypt's version of `bn_rsample` algorithm. The module also includes all functions which were used in the implementation of Jasmin's `bn_rsample`, namely, `bn_set0`, `bn_copy`, and `bn_subc`. The result of this compilation can be found in the accompanying code in file `W64_RejectionSamplingExtract.ec`.

Due to the fact that Jasmin's `bn_rsample` implements a special case of rejection sampling, we found that it was easy to relate the "high-level" EasyCrypt implementation `RS.rsample` to the Jasmin's "low-level" extract `XtrI.bn_rsample`. More specifically, we use EasyCrypt's pRHL to relate `XtrI.bn_rsample` with `RS.rsample` as follows:

```
lemma bn_rsample_spec m (a y : W64xN.t):
 let P = λ x. x < [a] in
  Pr[out ← RS.rsample(P)@m: out = y]
   = Pr[out ← XtrI.bn_rsample(a)@m: [out] = y].
```

Here, `W64xN.t` stands for the type of an array of size `nlimbs` of 64-bit binary words (i.e., `Array32.t W64.t`). To simplify the presentation we write `[x]` to denote a sequence of bits converted to unsigned integer (in EC this is done by using function `W64xN.bn`).

As a consequence of `bn_rsample_spec` and `rsample_pmf` we can immediately conclude the correctness of Jasmin's `bn_rsample`:

```
lemma bn_rsample_pmf m (a y: W64xN.t):
 0 ≤ [y] < [a]
⇒ Pr[out ← XtrI.bn_rsample(a)@m: out = y]
    = 1/[a].
```

In the next sections we address leakage-freeness of `bn_rsample`.

4.3 Derivation of LFdef(bn_rsample)

In the previous section we discussed the correctness of implementation of `bn_rsample` in Jasmin. In this section we address its leakage-freeness (more specifically, LFdef property). To do that, we compile Jasmin implementation to an EasyCrypt module with leakage annotations. The result is as follows:[7]

```
module XtrR(SC:Syscall_t) = {
 var leakages : leakages_t

 proc bn_rsample_i(a:W64xN.t): (int × W64xN.t) = {
   var q p i aux;
   p ← witness;
   q ← witness;
   i ← 0;
   leakages ← LeakAddr [] :: leakages;
   p ←@ bn_set0(p);

   leakages ← LeakAddr [] :: leakages;
   cf ← false;

   leakages ← LeakCond(!cf)
        :: LeakAddr [] :: leakages;
   while (!cf) {
     leakages ← LeakAddr [] :: leakages;
     aux ←@ SC.randombytes_32(
                        init_array nlimbs 64);
     p ← (Array32.init (λ i_0 ⇒ get64
           (WArray256.init8
            (λ i_0 ⇒ aux.[i_0])) i_0));

     leakages ← LeakAddr [] :: leakages;
     q ←@ bn_copy(p);

     leakages ← LeakAddr [] :: leakages;
     (cf, q) ←@ bn_subc(q, a);
     i ← i + 1;
     leakages ← LeakCond(!cf)
                  :: LeakAddr [] :: leakages;
   }
   return (i, p);
 }

 // includes leakage-annotated bn_subc/copy, etc.
}.
```

Recall that in the implementation of $\texttt{bn\_rsample}_i$ the counter `i` is a "logical" variable which we will use to derive properties.

The module `XtrR` also includes leakage-annotated versions of `bn_subc`, `bn_copy`, and `bn_set0` which we skip here for brevity. Our formalization contains proofs that these auxiliary functions are correct and constant-time (i.e., CTdef).

The analysis of leakage-freeness of $\texttt{bn\_rsample}_i$ is unusual because even if we proved that it terminates with probability 1 then we do not know in advance for how many iterations will it run. As a result, the contents of `XtrR.leakages` accumulator is probabilistic and depends on the number of iterations.

In the first step of our analysis we derive the probability of $\texttt{bn\_rsample}_i$ running for exactly `i` iterations and returning a specific element `x`. The proof is by induction on the number of iterations `i`.

```
op fail_once (a : int) : real = μ [0..2^nlimbs*64 -1]
                                  (λ x ⇒ a ≤ x).

lemma bn_rsample_pr m a i y: let t = 2^nlimbs*64 in
 1 ≤ i ⇒ 0 ≤ [x] < [a]
⇒ Pr[(c,x) ← XtrR.bn_rsample_i(a)@m
                          : c = i ∧ x = y]
```

[7] For the sake of clarity of presentation we clean the extracted EasyCrypt code and remove automatically generated boilerplate such as auxiliary variables and extra assignments.

```
    = (fail_once [a])^(i-1) / t.
```

Here, `(fail_once [a])` denotes the probability of failure of a loop iteration in `bn_rsample` which equals to the probability of uniformly sampling an element which is larger or equal than `[a]` from interval `[0..2^nlimbs*64 -1]`.

In the second step we prove that the contents of the leakage accumulator is in the functional relation with the number of iterations of the while-loop. More specifically, we define a function `samp_t` and establish that after termination of $\texttt{XtrR.bn\_rsample}_i$ the contents of `XtrR.leakages` equals to `samp_t i`. Intuitively, this shows that the leakages do not depend on the input arguments. At the same time, it does not mean that the result of the computation is independent of leakages.

```
op samp_t i =
 let p = [ LeakAddr []; ...] ++ set0_L ++ [...] in
 let s = [ LeakAddr []; ...] ++ copy_L ++ [...] in
 let loop j = repeat (j-1) [ LeakAddr []; ...] in
    p ++ loop i ++ s.
```

The constant `p` equals leakages before the while loop (here `set0_L` is a constant corresponding to leakages of `bn_set0` function). The constant `s` corresponds to the last iteration of while loop (here, `copy_L` corresponds to the leakages produced by a `bn_copy` procedure). And `(loop i)` corresponds to the first `i-1` iterations of the loop. It is important to understand that `sampt_t` is a non-probabilistic pure function which computes leakages only based on its arguments.

We show that function `samp_t` correctly captures the contents of `XtrR.leakages` by proving that the probability of `XtrR.leakages` being equal to a given list `l` is equal to the probability of geting leakage `samp_t i`:

```
lemma samp_t_correct a y l s m:
 XtrR.leakages{m} = s
⇒ Pr[(_,x)← XtrR.bn_rsample_i(a)@m:
                XtrR.leakages = l ++ s ∧ x = y]
   = Pr[(i,x)← XtrR.bn_rsample_i(a)@m:
                        samp_t i = l ∧ x = y].
```

Next, we observe that function `samp_t` is injective and therefore we can express the number of iterations `i` as an inverse of the leakages (if `l` is not in the image of `samp_t` then the inverse returns value `-1`):

```
lemma bn_rsample_leakf a y l s m:
 XtrR.leakages{m} = s
 ⇒ Pr[(_,x)← XtrR.bn_rsample_i(a)@m:
             XtrR.leakages = l ++ s ∧ x = y]
    = Pr[(i,x)← XtrR.bn_rsample_i(a)@m:
                 i = inv samp_t l ∧ x = y].
```

If we combine `bn_rsample_leakf` with `bn_rsample_pr` then we get the formula for the probability of producing list `l` and outputting the element `x`:

```
lemma bn_rsample_v a y l s m: XtrR.leakages{m} = s
⇒ let t = 2^nlimbs*64, i = inv samp_t l in
 Pr[(_,x)← XtrR.bn_rsample_i(a)@m:
           XtrR.leakages = l ++ s ∧ x = y]
  = if i ≤ 0 then 0 else (fail_once [a])^(i-1)/t.
```

Finally, by combining `bn_rsample_v` with `bn_rsample_pmf` we can derive that `bn_rsample` is leakage-free with respect to public input `a` (see Definition 3.2). In particular, we define a function `bn_rsample_f(a,l)` which calculates the conditional probability of generating leakages `l` with the public input `a` for any element `x` returned by `bn_rsample` (note that `bn_rsample_f` does not depend on `x`):

```
op bn_rsample_f(a,l) = let i = inv samp_t l in
                       let t = 2^nlimbs*64 in
  if i ≤ 0 then 0
   else (fail_once [a])^(i-1)*([a]/t).

lemma bn_rsample_leakfree m y a l s:
   XtrR.leakages{m} = s ⇒
 let v = Pr[x ← XtrR.bn_rsample(a)@m:
            XtrR.leakages = l ++ s ∧ x = y] in
 let w = Pr[x ← XtrR.bn_rsample(a)@m: x = y] in
  0 < w  ⇒  v/w = bn_rsample_f(a,l).
```

The function `bn_rsample_f` computes the inverse of `samp_t` on list `l` which is denoted by `i`. If `i` is larger than zero then we know that it would take exactly `i` iterations to produce leakages `l` (i.e., `XtrR.leakages = l`) and therefore we return probability which corresponds to `bn_rsample` running for exactly `i` iterations. In other case (i.e., `i ≤ 0`) the list `l` is not in the image of `samp_t` and, therefore, the probability of generating leakages `l` is 0.

To sum up, we have shown that Jasmin's `bn_rsample` procedure is correct (lemma `bn_rsample_pmf`) and leakage-free (lemma `bn_rsample_leakfree`).

4.4 pRHL proof of LF(bn_rsample)

In the previous section we illustrated a proof of leakage-freeness of `bn_rsample` by explicitly defining a leakage-function `samp_t` and then proving that leakages and output are independent. The main motivation for characterizing leakage-freeness directly in pRHL is to avoid the explicit handling of leakage (i.e., definition of function `samp_t`).

We now show how it can be achieved in the case of rejection sampling. The formalization relies on the framework presented in Section 3.2 and instantiating it for the respective functions.

At a very high level, the essence of the proof of the $\mathsf{LF}(f)$ equivalence is to decouple the computation of leakage and result in `XtrR.f`. This is a non-trivial task in challenging cases like rejection sampling where running time (and, therefore, leakages) are probabilistic. The strategy taken can be summarized in the following steps:

1. Exploit the LF equivalence and functional correctness of the called functions to simplify the code of `XtrR.bn_rsample` function;
2. Decouple the output from the leakages;
3. Restructure the rejection-loop to delay the sampling of the output.

Let us briefly overview what encompasses each of these steps. In the first step, the aim is to simplify `XtrR.bn_rsample`. To this end, one rewrites the LF equivalences for each called function, and replaces each Jasmin instruction by the corresponding semantics (given by correctness lemma). It leads to a program whose semantics is identical to that of `XtrI.bn_rsample`, but intertwined with code that accumulates leakages and values that are later discharged. For the `bn_rsample` case, we obtain something similar to:

```
a ←$ [0..2^(64·nlimbs)-1];
b ← a < [bnd];

[... leakage accumulation (including "b")]

while (!b) {
  a ←$ [0..2^(64·nlimbs) − 1];
  b ← a < [bnd];

  [... leakage accumulation (including "b")]
}

return a;
```

In the next step we focus on the sequence of the sampling of the result `a` and the evaluation of the acceptance criterion `b`. More generally, given a distribution over type `t` (`d: t distr`), and a predicate `P: t → bool`, we want to rewrite along the following equivalence:

$$\left\{\begin{array}{l} a \xleftarrow{\$} d; \\ b \leftarrow P\ a; \end{array}\right\} \sim \left\{\begin{array}{l} b \xleftarrow{\$} \text{dbiased } (\mu\ d\ P); \\ a \xleftarrow{\$} \text{if } b \text{ then dcond } d\ P \\ \qquad\qquad \text{else dcond } d\ (\text{predC } P); \end{array}\right\}$$
$$: \text{true} \Longrightarrow = \{a, b\}$$

Where `dbiased p` is the Bernoulli distribution with parameter `p`, `dcond d Ev` is the conditional probability of `d` given `Ev`, and `predC P` is the complement of the predicate `P`. Notice that on the right-hand side we sample the value `b` from Bernoulli distribution in a manner which does not depend on the variable `a`. Later this will allow us to delay the sampling of the result (i.e., value of `a`). In our formalization we define an EC theory that proves the above equivalence generically and later we instantiate it for the case of rejection sampling.

The final step reshapes the loop structure to move the sampling of `a` outside of the while-loop. Again, we defined an EC theory to give a generic and reusable implementation. In particular we define a module type `AdvLoop` which represents an arbitrary computation which is not essential for the restructuring of the loop.

Next, the module `RejLoop` (which is parameterized by `AdvLoop` module) implements functions `loopEager` and `loopLazy`. The difference between these two functions is that in `loopEager` we sample value `a` at each iteration of the while-loop and in `loopLazy` we sample `a` only once after the while-loop is terminated.

```
abstract theory RejectionLoop.

type t.

op dt: t distr.
op p : t → t → bool.


module type AdvLoop = {
  proc loop_init(b: bool): unit
  proc loop_body(b: bool): unit
}.

module RejLoop(L:AdvLoop) = {
  proc loopEager (bnd: t) = {
    var a, b;
    b ←$ dbiased (μ dt (p bnd));
    a ←$ if b then dcond dt (p bnd)
            else dcond dt (predC (p bnd));
    L.loop_init(b);
    while (! b) {
      b ←$ dbiased (μ dt (p bnd));
      a ←$ if b then dcond dt (p bnd)
              else dcond dt (predC (p bnd));
      L.loop_body(b);
    }
    return a;
  }

  proc loopLazy (bnd: t) = {
    var a, b;
    b ←$ dbiased (μ dt (p bnd));
    L.loop_init(b);
    while (!b) {
      b ←$ dbiased (μ dt (p bnd));
      L.loop_body(b);
    }
    a ←$ dcond dt (p bnd);
    return a;
  }
}.


[...properties...]

end RejectionLoop.
```

Using probabilistic relational Hoare logic we prove that `loopLazy` and `loopEager` are equivalent:

```
equiv rejloop_eq (L <: AdvLoop):
 RejLoop(L).loopEager ~ RejLoop(L).loopLazy
 : ={bnd, G_L} => ={res, G_L}.
```

The proof of the above property relies on the formalization of the equivalence of Eager and Lazy random oracles from the EC's standard library (`PROM.ec`).

After applying `rejloop_eq` to rejection sampling loop we arrive at the program where acceptance criteria and leakage computations are not intertwined with the output sampling. This allows us to easily conclude LF(`bn_rsample`) equivalence because the probabilistic leakage accumulation and the sampled output become fully decoupled.

4.5 Comparing both proof strategies

When proving leakage freedom using the pRHL characterization, one can argue that the reasoning is more code-oriented — in essence, it applies transformations to the program in order to slice it in two independent paths (leakage and secret output generation). Even if these transformations are generic and can conceivably be applied to similar code patterns, the choice of which transformations suit each concrete example critically depends on fully understanding the leakage model. To that extent, the insight obtained by tackling the problem directly with Definition 3.2 could provide hints on what manipulations are needed. In any case, and after the fact, the pRHL proof will arguably be more elegant as they abstract the leakage internals.

On the other hand, the full proof based on Definition 3.2 is significantly shorter as it directly addresses the property under consideration. The proof size of both strategies will be on par (approx. 450 vs. 400 loc) only if we exclude the proofs concerning reusable components of the first strategy[8].

5 Conclusions

In this work we studied leakage-freeness of probabilistic Jasmin programs with secret outputs. We motivated our work by explaining that the "constant-time" property associated with deterministic programs fails for the probabilistic case with secret outputs. We proposed novel definition of leakage-freeness and provided the semantical and pRHL characterizations. We proved that these are equivalent, composable, and generalize the "constant-time" criteria used for deterministic programs. Also we illustrated the derivation of leakage-freeness for rejection sampling algorithm which has probabilistic runtime. To the best of our knowledge, the leakage-freeness for probabilistic programs have not yet been addressed in theorem provers.

[8]Approx. 270loc for instantiating the generic formalized framework; 300loc for the proofs in the abstract theories of the two employed transformations; and 730loc for the proofs of the bignum library

A final word on the role of termination in the assessment of leakage-freeness. As pointed out in [9], termination-insensitive non-interference might fail to prevent the disclosure of secrets in the presence of divergence behavior. In this work, we choose to adhere to the assumption that termination is independently checked as part of a safety analysis of the source program (c.f. [5]). However, we recognize that examples such as rejection sampling addressed in this paper call for relaxing that assumption. For that reason, in our formalization, we have relaxed that constraint by allowing possibly divergent programs when assessing the independence of outputs and leakage (under a slight generalization of the Definition 3.4). But lifting it to a sensible notion of leakage-freeness of possibly divergent programs would force us to move to a termination-sensitive characterization of non-interference in EC, which we left as future work.

Acknowledgements

José Bacelar Almeida was supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020 (DOI 10.54499/LA/P/0063/2020). Denis Firsov was supported by the Estonian Research Council grant PSG749. Dominique Unruh was supported by ERC consolidator grant CerQuS (Certified Quantum Security, 819317), Estonian Centre of Excellence in IT (EXCITE, TK148), Estonian Centre of Excellence "Foundations of the Universe" (TK202), and Estonian Research Council PRG grant "Secure Quantum Technology" (PRG946).

References

[1] Accompanying EasyCrypt development. https://github.com/dfirsov/jasmin-leakage-freeness. Accessed: 2025-12-03.

[2] Archived accompanying EasyCrypt development. https://doi.org/10.5281/zenodo.14281008. Accessed: 2025-12-05.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[4] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1622, 2019.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[6] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked ZKP for NP relations: Formally verified security proofs and implementations of MPC-in-the-head. In *Proceedings of*

the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2587–2600, 2021.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber episode IV: implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3): 164–193, 2023. doi: 10.46586/TCHES.V2023.I3.164-193.

[8] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber - episode V: machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part II*, volume 14921 of *Lecture Notes in Computer Science*, pages 384–421. Springer, 2024. doi: 10.1007/978-3-031-68379-4_12.

[9] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 333–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-88313-5.

[10] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer, 2011.

[11] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, pages 146–166. Springer, 2013.

[12] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving uniformity and independence by self-composition and coupling. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 385–403. EasyChair, 2017. doi: 10.29007/VZ48.

[13] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 462–476. ACM, 2021. doi: 10.1145/3460120.3484761.

[14] Tom Chothia, Yusuke Kawamoto, Chris Novakovic, and David Parker. Probabilistic point-to-point information leakage. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 193–205, 2013. doi: 10.1109/CSF.2013.20.

[15] NIST Computer Security Division. ML-DSA: Module-Lattice-Based Digital Signature Standard. FIPS Publication 204, National Institute of Standards and Technology, U.S. Department of Commerce, Aug 2024. URL https://csrc.nist.gov/pubs/fips/204/final.

[16] Denis Firsov and Dominique Unruh. Reflection, rewinding, and coin-toss in EasyCrypt. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–179, 2022.

[17] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi: 10.1109/JSAC.2002.806121.

[18] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 83–96. ACM, 2022. doi: 10.1145/3548606.3560689.

[19] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1094–1111. IEEE, 2023. doi: 10.1109/SP46215.2023.10179418.