Leakage-Free Probabilistic Jasmin Programs

José Bacelar Almeida
Universidade do Minho
Braga, Portugal
INESC TEC, Porto, Portugal
jba@di.uminho.pt

Denis Firsov
Guardtime
Tallinn University of Technology
Tallinn, Estonia
denis@cs.ioc.ee

Tiago Oliveira
MPI-SP
Bochum, Germany
tiago.oliveira@mpi-sp.org

Dominique Unruh
University of Tartu
Tartu, Estonia
unruh@ut.ee

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

To test our definitions we use Jasmin toolchain to develop a rejection sampling algorithm and prove (in EasyCrypt) that the implementation is leakage-free whilst not being constant-time.

***Index Terms*—cryptography, formal methods, EasyCrypt, leakage-freeness, side-channels, timing attack, rejection sampling, Jasmin**

I. 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 [1] and Jasmin [2] 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 assembler-like language and a compiler. In Jasmin, we can write a highly optimized implementation of some cryptographic function, and have it compiled to actual assembler (for various platforms such as x86-64). In addition, Jasmin produces EasyCrypt code that is guaranteed[2] to be functionally equivalent to the generated assembler code! This allows us to do cryptographic security proofs in EasyCrypt, and know that they also apply to the assembler implementation (which hopefully is the one actually used in the end).

But Jasmin goes further than that: The exported EasyCrypt code contains instructions that explicitly describe side-channel leakage that can happen in the assembler code (e.g., timing leakage). Then, again in EasyCrypt, we can prove that the leakage does not depend on the secret inputs and that guarantee then carries over to the assembler code. The current released versions of Jasmin aims at timing attacks in what they call the "baseline model" in which control flow (i.e., the program counter) and the addresses of memory accesses are leaked. Also, there exist development branches of the Jasmin compiler which support other leakage models (e.g., leaking the cache line, and variable time assembler instructions) [3]. Unfortunately, these branches of Jasmin does support instructions for random byte sampling which are necessary for our goal of investigating leakage-freenes for probabilistic Jasmin programs.

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 leakage.

Previously all Jasmin programs were deterministic. For this case, the Jasmin approach for showing leakage-freeness through timing side-channels was to prove, essentially, that the code is constant-time (leakage only depends on the public inputs).

Recently, Jasmin was extended with primitive which generates random bytes (`#randombytes` function). Therefore, now the runtime may well depend on the random choices (not constant-time) but still not leak anything about any secrets. In this work, we propose new definition of leakage-freeness for

[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.

[2] Of course, we again need to assume that Jasmin itself does not contain bugs here.

probabilistic Jasmin programs. We also show how it relates to the definition of leakage-freeness for deterministic programs, explain why the old definition fails for the probabilistic case, and most importantly prove that the new definition is compositional.

To motivate and test our new definition we implement and prove leakage-freeness for rejection sampling algorithm. In cryptographic protocols, we often need to sample numbers at random. However, low-level random number generators usually provide only a sequence of random bits (or bytes). This can be interpreted as a random number from $\{0, \ldots, 2^\ell - 1\}$. Unfortunately, this does not give us all distributions: for example, if we would like to sample from $\{0, \ldots, p-1\}$, where p is a prime (and thus not a power of two). This problem can be resolved by rejection sampling: For some ℓ with $2^\ell \geq p$, sample repeatedly from $\{0, \ldots, 2^\ell - 1\}$ until you get a value $< p$.

The downside of rejection sampling is that it does not have an apriori termination time which means that we do not know how long will it take to produce an element which satisfies the criteria.

We implement rejection sampling in Jasmin, and prove (in EasyCrypt) that rejection sampling always returns uniform element within the desired range. More interestingly, we show (in EasyCrypt) that the Jasmin implementation is indeed leakage-free. Since the running time of the rejection sampling is randomized, and since it is not possible to mask it by upper bounding the running time, we cannot show that it is constant-time (the usual approach of showing leakage-freeness in Jasmin) but use our new relaxed criterion instead.

Contributions: Our technical contributions include the following results:

- We introduce workflow of Jasmin workbench and explain motivation behind our work in Sec. II-B.
- We give semantic and pRHL characterization of leakage-freeness for probabilistic Jasmin programs (Sec. III-B).
- We prove equivalence and compositionality of our leakage-freeness characterizations and relate them to constant-time definition from prior work (Sec. III-C).
- We implement a generic rejection sampling in EasyCrypt with proofs of its correctness and termination (Sec. IV-A).
- We implement uniform sampling in Jasmin as a special case of rejection sampling (Sec. IV-B).
- We present direct (semantical) and pRHL derivations of leakages-freeness for Jasmin's uniform sampling (Sec. IV-C and Sec. IV-D).

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 a supplementary material.

A. Related Work

The Jasmin toolchain was introduced with Coq proofs of the correctness of the compiler in [2]; it was connected to EasyCrypt in [4]; the toolchain was extended to cover leakage-freeness guarantees in [3], [5]. Several cryptographic schemes have been implemented in Jasmin: the ChaCha20 streamcipher, the Poly1305 and Gimli hash function (all in [4]), the scalar multiplication algorithm for the elliptic curve Curve25519 in [2], the SHA-3 hash function in [6], the Kyber public-key encryption scheme in [7], and the MPC-in-the-head protocol in [8]. Of these, most contain proofs of functional correctness and derivation of constant-time property for deterministic programs.

The recent work [7] analyses various Kyber [9] implementations in Jasmin and derives their functional correctness (i.e., that they match the abstract specification in EasyCrypt). At the moment the published preprint only briefly mentions constant-time property. It does not provide any information on how "constant-time" is defined or derived. We believe that our new definitions of leakage-freeness would come in handy especially because Kyber is a probabilistic algorithm which also makes use of rejection sampling. Also, their rejection sampling is a routine from the Kyber standard for sampling elements from a specific ring, and functional correctness (i.e., uniformity) is not shown for the actually implemented sampling but an idealized version of it (with hash functions replaced by fresh random values). This is a necessary consequence of the fact that the Kyber specifications prescribe a very specific sampling algorithm that simply happens not to be exactly uniform, and also probably hard to prove even approximately uniform outside the random oracle model. In our work, Jasmin implementation of rejection sampling is provably uniform and additionally, we specify and obtain leakage-freeness for rejection sampling.

II. Preliminaries

A. 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 [10].

To our readers who are not familiar with EasyCrypt also suggest to read a short EasyCrypt introduction in [11, Section 2]. More information on EasyCrypt can be found in the EasyCrypt tutorial [10].

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.

B. Jasmin Workbench

Jasmin is a toolchain for high-assurance and high-speed cryptography [2]. 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.

1) Jasmin Basics: We explain the basics and workflow of Jasmin development on a simple example. More specifically, our goal is to implement a procedure which with equal probabilities returns values `0` or `1` (encoded as bytes). Below is the "naive" implementation of such a program in Jasmin:

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

   _byte_p = byte_p;
   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 allocated in the 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. The variable `_byte_p` has type `reg ptr u8[1]` which indicates that it uses a register to store a pointer to value `u8[1]`. Next, we store a pointer to `byte_p` in `_byte_p`. Next, we generate a random byte with a systemcall `#randombytes`. The systemcall takes a pointer to 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 `XtrI` with a 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, _byte_p:W8.t Array1.t;
   _byte_p ← witness;
   byte_p  ← witness;
   _byte_p ← byte_p;
   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]` and a pointer to such array `reg ptr u8[1]` were both translated to `W8.t Array1.t`. EasyCrypt model does not recognize a difference between values allocated on stack and in registers, so this information is abstracted away during translation.

Since `random_bit_naive` uses a systemcall `#randombytes` then Jasmin generates a module `XtrI` which is parameterized by a "provider" of systemcalls `SC`. In our example, the systemcall `#randombytes` is translated to an invocation of $\texttt{SC.randombytes}_1$ procedure. Clearly, that such interpretation of systemcalls makes it harder to rigorously define the semantics of Jasmin programs, but at the same time it allows users to choose their own interpretation of systemcalls. Also, Jasmin produces a module $\texttt{SC}_\texttt{D}$ with the "default" interpretation of systemcalls. In our work we use the default interpretation (i.e., $\texttt{SC}_\texttt{D}$ systemcall provider) which models `#randombytes` as a generator of truly random bytes. Alternatively, one could interpret `#randombytes` as an invocation of pseudo-random generator.

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.[3]

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 (secret) inputs and the output. To perform leakage-free analysis a user can instruct Jasmin compiler to extract a program to EasyCrypt with leakage annotations (leakage annotations are added automatically by Jasmin). In this case, the resulting EasyCrypt module `XtrR` has a global variable `leakages` which is used in the extracted EasyCrypt procedures to accumulate information which can get leaked in case of a timing attack. For example, if we compile `random_bit_naive` to EasyCrypt with leakage annotations then the result is as follows:

```
module XtrR(SC:Syscall_t) = {
 var leakages : leakages_t // global variable
 proc random_bit_naive () : W8.t = {
   var r, aux0: W8.t;
   var aux, byte_p, _byte_p:W8.t Array1.t;
   _byte_p ← witness;
   byte_p ← witness;
   leakages ← LeakAddr([]) :: leakages;
   aux ← byte_p;
   _byte_p ← aux;
   leakages ← LeakAddr([]) :: leakages;
   aux ←@ SC.randombytes1 (_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;
     aux0 ← (W8.of_int 0);
     r ← aux0;
   } else {
     leakages ← LeakAddr([]) :: leakages;
     aux0 ← (W8.of_int 1);
     r ← aux0;
   }
   return (r);
 }
}.
```

The entries in the `leakages` accumulator must be understood as a data which an attacker could learn if they would carry-out a timing attack. The leakage annotations are added for every basic statement of the Jasmin program.

Observe that in procedure `XtrR.random_bit_naive` the call to $\texttt{SC.randombytes}_1$ only adds a leakage value `LeakAddr []` to the `leakages` accumulator. This means that this operation does not leak any information about result of its computation. At the same time, since its execution requires time then this is modelled by adding an empty leakage value `LeakAddr []`.

Also notice that `if`-statements leak the boolean value of the conditional statement. As a result the boolean value `byte_p.[0] < W8.of_int 128` is added to the accumulator. This indicates that a timing attack might reveal which branch of the `if`-statement was executed. As a result, we can say that the current implementation of `random_bit_naive` is not leakage-free since it leaks some data about the actual dataflow of the program execution. In this particular case, the function `random_bit_naive` entirely leaks its output.

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 ptr u8[1] _byte_p;
   reg u8 r;
   _byte_p = byte_p;
   byte_p = #randombytes(_byte_p);
   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. We can prove that the new version of `random_bit` is a uniform distribution of values `0` and `1`. However, the more interesting aspect is whether the 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, aux0: W8.t;
   var aux, byte_p, _byte_p: W8.t Array1.t;
   _byte_p ← witness;
   byte_p ← witness;

   leakages ← LeakAddr([]) :: leakages;
   aux ← byte_p;
   _byte_p ← aux;
   leakages ← LeakAddr([]) :: leakages;
   aux ←@ SC.randombytes1 (_byte_p);
   byte_p ← aux;
   leakages ← LeakAddr([0]) :: leakages;
   aux0 ← byte_p.[0];
   r ← aux0;
   leakages ← LeakAddr([]) :: leakages;
   aux0 ← (r `&` (W8.of_int 1));
   r ← aux0;
   return (r);
 }
}.
```

Now it must be easy to see that after execution of `random_bit` function the `leakages` accumulator does not contain any data specific to the output of the program. Moreover, the same list of leakages is generated on every execution of the `random_bit` function (i.e., the resulting `XtrR.leakages` is deterministic and not probabilistic). Therefore, we can "intuitively" conclude that `random_bit` is leakage-free.

However, to be able to argue about leakage-freeness formally we must give rigorous definitions of leakage-freeness and cryptographic constant-time (see Sec. III).

[3]In our work we do not actually derive properties about the function `random_bit_naive` because as we will see later it is not leakage-free, instead we develop a function `random_bit` for which we derive correctness and leakage-freeness.

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

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

Here, $=\{res\}$ denotes the equality of outputs of the left (`XtrI.f`) and the right (`XtrR.f`) programs. Informally, this equivalence asserts that we obtain equally distributed results when running both programs in initial memories that equate the values of the input arguments `pin` and `sin`. The property can be easily confirmed in EC for concrete instances, as it can typically be proved automatically resorting to EC's `sim` tactic.

Before Jasmin was extended with `#randombytes` primitive all its programs were deterministic. In this case, proving that program is leakage-free (or "constant-time" in the parlance of the prior work) requires only to prove that probability of producing a particular leakages 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 ].
```

Unfortunately, this definition fails to capture leakage-freeness for probabilistic programs. For example, one could easily prove that `random_bit_naive` program (see Sec. II-B2) satisfies the Definition 3.1 from above. This happens because the output of `random_bit_naive` which must stay secret is generated by sampling and does not depend on the input arguments.

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

A. 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 3.2 (Leakage-Free Jasmin Programs): Let `f` be a total Jasmin program 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 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 "leakage" distribution does not depend on the secret input `sin` and 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 understand that computations themselves (i.e., function `XtrR.f`) cannot introspect (i.e., analyze) leakages in `XtrR.leakages`.

To apply this definition to `random_bit` function defined in Sec. II-B2 we must implement a function which computes the conditional probability described above. For the `random_bit` function it looks as follows:

```
op g l = let random_bit_l
           = [LeakAddr  []; LeakAddr [0];
              LeakAddr []; 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. By using the 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 because the "leakage" and output distributions are not independent.

B. 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 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 $\mathsf{CT}(f)$) 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. In essence, the given equivalence enforces that, running two copies of the program `f` (instrumented with leakage accumulator) in a pre-state that equates only their public inputs and the initial leakage, shall produce equal leakages on the post-state. Thus, the distribution of leakage to be independent on secret inputs. Moreover, the above pRHL definition is extremely useful, as is often automatically proved through the EC's `sim` tactic, and moreover it also enjoys nice properties such as compositionality.

For exactly the same reasons as already explained for Definition 3.1, the Definition 3.3 does not enforce independence of the leakages with respect to secret outputs, turning it useless for capturing leakage-freeness for probabilistic programs.

Nonetheless, there is a natural generalization of the constant-time property (i.e., Definition 3.3) for probabilistic programs, by asking the leakage to be simulated without access to secret inputs.[4]

*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 hold: `∀ pin sin sin',`

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

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 EC, for simple programs where runtime is not probabilistic the derivation of LF property can be done in only couple of lines of code. For example, for `random_bit` case after we instantiate 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 simply performs a symbolic simulation.

In contrast, the direct derivation of Definition 3.2 requires us to specify explicit function for calculating leakages and make a careful analysis of conditional probabilities.

In Sec. IV we perform a much more challenging analysis of leakage-freeness for rejection sampling algorithm where runtime is probabilistic.

C. Properties

We collect now main properties relating the various definitions. They have been fully formalized in EC[5].

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

1) $\mathsf{LF}(f) \Longrightarrow \mathsf{CT}(f)$
2) $\mathsf{det}(f) \Rightarrow (\mathsf{LF}(f) \Longleftrightarrow \mathsf{CT}(f))$
3) $\mathsf{LF}(f) \Longleftrightarrow \mathsf{LFdef}(f)$
4) $\mathsf{CTdef}(f) \Longleftrightarrow \mathsf{CT}(f)$

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

```
∃ f_spec, ∀ p s,
  hoare[XtrI.f: pin=p ∧ sin=s ⇒ res = f_spec(p,s)].
```

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 [11] 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.
```

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

[4]In order to better understand the role of termination in the results of interest, we choose not to to assume totality beforehand in our formalization. Interestingly, the generality of results do carry over arbitrary (possibly divergent) procedures, under a slight generalisation of the LF equivalence. Unfortunately, that broaden scope clearly leads us beyond the expressiveness of the underlying model, making it unclear how (and if) such a generalisation relates a sensible notion of leakage-freeness.

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

any subsets of leakage traces lie in the unit interval). Moreover, the associated condition enforces the equality of distributions

$$\texttt{dR} = \texttt{dI} \ \texttt{`*`} \ \texttt{dLeak},$$

where `*` denotes the product of 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 `sout`;
- both are leakage-free (i.e. $\mathsf{LF}(f)$ and $\mathsf{LF}(g)$ holds).

Then, the program

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

is itself leakage-free $\mathsf{LF}(h)$.
Its proof relies on the observation that `XtrI.f` and `XtrI.g`, being stateless, can be moved 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.

IV. Rejection Sampling

In Jasmin we can use `#randombytes` systemcall to generate bytes uniformly at random. However, this does not immediately give us uniform distributions on sets whose cardinality is not power of 2. In this section our goal is to describe 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 will it 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, but the runtime is probabilistic.

We discovered that the standard library of EasyCrypt has a formalization of rejection sampling algorithm in theory `Dexpected.ec`. However 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. The formalization in EasyCrypt's standard library is based on the game rewriting approach.

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.

A. Rejection Sampling in EasyCrypt

We start by implementing a rejection sampling algorithm in EasyCrypt. Our algorithm is parameterized by a lossless distribution `d` of parameter type `X`. 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.
```

B. 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. IV-A)[6]. 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` systemcall 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.[7]

Now we describe an implementation of a Jasmin program `bn_rsample`$_i$`(a)` (prefix `bn` stands for big-number) whose input `a` is an `nlimb`-array representing a number from the interval $[0...2^{64\cdot \texttt{nlimbs}} - 1]$ 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 systemcall `#randombytes` to sample a random number `p` from the interval $[0...2^{64\cdot \texttt{nlimbs}}-1]$. Then we subtract `p` from `a` by using a `bn_subc` function.[8] 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 additional 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();
  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;
```

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

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

[8]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 the EasyCrypt's pRHL to relate that `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`.

C. *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:[9]

```
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]
     = (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 $[0..2^{\texttt{nlimbs}*64}-1]$.

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.

[9]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.

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

The constant `prefix` equals to leakages before the while loop (here `set0_L` is a constant corresponding to leakages of `bn_set0` function). The constant `suffix` 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.

We show that `samp_t` correctly captures the contents of `XtrR.leakages` by proving that the probability of `XtrR.leakages` being equal to a list `l` equals to the probability of `(samp_t i)` being equal to `l`:

```
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 returns the conditional probability of generating leakages `l` with the public input `a` given that the procedure `bn_rsample` returned an element `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., $i \leq 0$) 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`).

D. pRHL *proof of* LF(*bn_rsample*)

In the previous section we proved 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 III-B and instantiating it for the respective functions.

At a very high level, the essence of the proof of the 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;
```

The next step we focus on the sequence of the sampling of the result `a` and the evaluation of the acceptance criteria `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:

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

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.

V. Conclusions

In this work we studied leakage-freeness of probabilistic Jasmin programs. We motivated our work by explaining that the "constant-time" property associated with deterministic programs fails for the probabilistic case. 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. 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.

References

[1] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Annual Cryptology Conference*. Springer, 2011, pp. 71–90.

[2] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

[3] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, and S. Priya, "Enforcing fine-grained constant-time policies," Cryptology ePrint Archive, Paper 2022/630, 2022, https://eprint.iacr.org/2022/630. [Online]. Available: https://eprint.iacr.org/2022/630

[4] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 965–982.

[5] G. Barthe, B. Gregoire, V. Laporte, and S. Priya, "Structured leakage and applications to cryptographic constant-time and cost," Cryptology ePrint Archive, Paper 2021/650, 2021, https://eprint.iacr.org/2021/650. [Online]. Available: https://eprint.iacr.org/2021/650

[6] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. 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*, 2019, pp. 1607–1622.

[7] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, V. Laporte, J.-C. Léchenet, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe *et al.*, "Formally verifying Kyber episode IV: Implementation correctness," *Cryptology ePrint Archive, Paper 2023/215*, 2023, https://eprint.iacr.org/2023/215. [Online]. Available: https://eprint.iacr.org/2023/215

[8] J. B. Almeida, M. Barbosa, M. L. Correia, K. Eldefrawy, S. Graham-Lengrand, H. Pacheco, and V. 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*, 2021, pp. 2587–2600.

[9] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: a cca-secure module-lattice-based kem," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 353–367.

[10] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of Security Analysis and Design VII*. Springer, 2013, pp. 146–166.

[11] D. Firsov and D. Unruh, "Reflection, rewinding, and coin-toss in easycrypt," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022, pp. 166–179.