Schnorr protocol in Jasmin

Denis Firsov [1 4], Tiago Oliveira [2], and Dominique Unruh [3]

[1]Tallinn University of Technology, Estonia
[2]Max Planck Institute for Security and Privacy, Bochum, Germany
[3]University of Tartu, Estonia
[4]Guardtime, Tallinn, Estonia

May 25, 2023

Abstract

We implement the Schnorr proof system in assembler via the Jasmin toolchain, and prove the security (proof-of-knowledge property) and the absence of leakage through timing side-channels of that implementation in EasyCrypt.

In order to do so, we show how leakage-freeness of Jasmin programs can be proven for probabilistic programs (that are not constant-time). We implement and verify algorithms for fast constant-time modular multiplication and exponentiation (using Barrett reduction and Montgomery ladder). We implement and verify the rejection sampling algorithm. And finally, we put it all together and show the security of the overall implementation (end-to-end verification) of the Schnorr protocol, by connecting our implementation to prior security analyses in EasyCrypt (Firsov, Unruh, CSF 2023).

Contents

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 [BGHB11] and Jasmin [ABB+17] frameworks aim to resolve this issue. EasyCrypt is a tool in which we can write cryptographic security proofs and verified 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

[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.

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) [SBG+22]. Unfortunately, these branches of Jasmin does support instructions for random byte sampling which are necessary for our project.

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

But we stress that this is not an automatic processes: For each protocol, considerable human effort is needed to write the proofs and implementations. And the proofs not only need to cover high-level security properties but also relate the high-level representation of the protocol with Jasmin's low-level code, and additionally prove the absence side-channel leakage. Because of this, there are only very few end-to-end analyses in Jasmin to date (see Sec. 1.1).

In this work, we extend this to cover another protocol: The Schnorr proof system [Sch90].

In a nutshell, for a fixed prime p, and group element $s \in \mathbb{Z}_p^*$ (and a fixed generator $g \in \mathbb{Z}_p^*$), this protocol allows us to prove that we know the discrete logarithm of s, i.e., some w such that $g^w = s$. It's message flow is:

$$
\begin{array}{lcl}
P & \xrightarrow{\qquad z := g^r \qquad} & V \\
 & \xleftarrow{\qquad \text{random } c \qquad} & \\
 & \xrightarrow{\qquad t := r + cw \qquad} & \text{Check: } g^t = zs^c
\end{array}
$$

The protocol is, for example, useful as an identification scheme, but also as a proof system, and as a building block for group-based signature schemes [Sch90]. But besides the intrinsic interest in this protocol, it helps us to better understand the Jasmin/EasyCrypt ecosystem and its power. Namely, while the protocol itself is quite simple, it has a number of features that push the boundary of what has been done in Jasmin so far:

- It is an *interactive* protocol (with two parties, one of them potentially corrupt).

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

- It involves randomness in a crucial way. It crucially relies on the unpredictability of the message from the other party.
- The security definition (proof of knowledge) is more advanced, involving not only an adversary, but existentially quantified entities such as an extractor.
- The security analysis relies on a technique that pushes the boundaries of EasyCrypt, namely "rewinding".[3]
- It involves modular arithmetic (in particular modular exponentiation). Modular exponentiation is extremely important in many cryptosystems, yet the gap in abstraction level between EasyCrypt's mathematical definition of the ring $\mathbb{Z}_p$, and an optimized constant-time implementation in Jasmin is very large. Verified and reusable optimized implementations of modular multiplication and exponentiation will benefit end-to-end verification of other protocols, too.

Our contribution. Our work on Schnorr's protocol lead to the following contributions:

Probabilistic leakage-freeness. The need for randomization in our work triggered the addition of a `randombytes` primitive in Jasmin, allowing to write protocols that use randomness. However, the existing approach in Jasmin for showing leakage-freeness is to show, essentially, that the code is constant-time (leakage only depends on the public inputs). Yet, in probabilistic programs, the runtime may well depend on the random choices (not constant-time) but still not leak anything about any secrets. Hence the existing approach can be too restrictive (and is for "rejection sampling", see below). We show how to model leakage-freeness for Jasmin programs in the probabilistic setting, and prove (for backwards compatibility and ease-of-use) that the more restrictive definition of constant-time implies the new leakage-freeness. (The latter we formalized in EasyCrypt.)

Rejection sampling. In Schnorr's protocol, we need to pick random numbers from $\{0, \ldots, p-1\}$. However, random number generators (such as `randombytes`) usually provide only a sequence of random bits (or bytes). This can be interpreted as a random number from $\{0, \ldots, 2^\ell - 1\}$ for some ℓ, but not $\{0, \ldots, p-1\}$. (As 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 from repeatedly from $\{0, \ldots, 2^\ell - 1\}$ until you get a value $< p$.

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,

[3] Rewinding has been covered in EasyCrypt already in [FU23]. However, we believe it is important to validate that this carries over to end-to-end verification in the Jasmin/EasyCrypt framework.

and since it is not possible to mask it by upper bounding the time,[4] we cannot show that it is constant-time (the usual approach of showing leakage-freeness in Jasmin) but use our new relaxed criterion instead.

Finally, we stress that we made sure that our rejection sampling algorithm and analysis is very generic: It is parameterized over an arbitrary predicate that describes what values are "valid", and returns a uniformly random "valid" value. (For Schnorr, this predicate is simply $P(x) := (x < p)$.)

Modular multiplication and Barrett reduction. The existing libjbn [lib] library for big numbers for Jasmin has support for some operations over big integers, but it does not include modular multiplication or modular exponentiation.

When speed matters, modular multiplication is nontrivial since the repeated computation of the modulo-operation will be a major performance bottleneck. A solution to this problem is the Barrett reduction [Bar87]: Here a slow precomputation is done once to derive a so-called Barrett factor. (Involving division, and depending only on the modulus n.) And using that Barrett factor, we can perform a reduction modulo n using only cheap operations such as additions/multiplications over integers. Based on this, we can then implement multiplication as an integer multiplication followed by a Barrett reduction.

We implement the Barrett reduction and modular multiplication algorithms in Jasmin, and prove their correctness and constant-time property in EasyCrypt. Our approach in this (and in the contributions below) is to develop both the low-level Jasmin code, as well as a simpler, less low-level EasyCrypt version, prove the properties of the latter, and show equivalence to the Jasmin code.

Modular exponentiation and Montgomery ladder. Once we have modular multiplication, we can implement modular exponentiation on top. The well-known square-and-multiply algorithm computes x^n in any monoid. However, square-and-multiply is not constant time: Depending on the Hamming weight of n, we may have more or less multiplications, leaking information about n. A variation of the square-and-multiply algorithm that avoids this, while keeping its high efficiency, is the Montgomery ladder [Mon87].

We implement the Montgomery ladder in Jasmin, using the multiplication based on Barrett reduction, and prove its correctness (that it actually computes x^n) and its constant-time property in EasyCrypt.

Our implementation and analysis the Montgomery ladder is generic. By replacing the modular multiplication algorithm by a constant-time multiplication algorithm in another monoid, one gets a constant-time exponentiation algorithm for that semigroup without redoing any proofs. For example, future work might use this to implement exponentiation in elliptic curves.

Schnorr protocol. We now have leakage-free algorithms for random sampling and modular exponentiation; we can implement the Schnorr protocol.

We implement the protocol in Jasmin. More precisely, we implement four Jasmin procedures for the prover's first message, the verifier's challenge genera-

[4] At least not if we want perfectly uniform numbers. In that case, no sampling algorithm with bounded running time exists that uses only random bits [KSU13].

tion, and the prover's second message, and the verifier's final decision whether it accepts. Note that Jasmin does not provide any mechanism to put them together as an interactive protocol. Instead, we write a C wrapper that invokes the compiled Jasmin procedures. This serves as a demonstration how the Jasmin procedure can be embedded in a larger application (say a network protocol). We prove each of them leakage-free by using the results from the previous contributions (in EasyCrypt).

The interesting part here is the security of the resulting code. We focus on the proof-of-knowledge property (a.k.a. knowledge soundness) here that informally says that a prover cannot succeed in Schnorr's protocol unless it actually knows a witness w with $s = g^w$. This security property was already formalized in [FU23] in EasyCrypt, and the security of the Schnorr protocol (in its abstract, far from low-level form) proven.

Here, we need to reformulate the security property for the Jasmin programs. (We cannot directly apply the existing definition since Jasmin programs cannot, for example, keep internal state.) What makes this security property interesting is that it is defined not only with respect to an all-quantified adversary breaking the protocol, but also an existentially quantified entity (the extractor) "helping" the protocol. In EasyCrypt, such entities are typically given as explicitly spelled out source code. So we end up with the interesting case we have three kinds of entities in our definition: (a) All-quantified adversaries who could be any unspecified EasyCrypt program. (b) Explicit protocol specification from Jasmin. (c) Explicit entities (extractor) written abstractly in EasyCrypt.[5] We believe that our solution also provides guidance for future proofs of other cryptographic properties for Jasmin programs that use a simulation-based paradigm.

Unfortunately, we were not able to reuse the security proofs of Schnorr protocol from [FU23] directly since it depends on the formalization of cyclic groups which vanished from the latest edition of EasyCrypt standard library. Instead we used the framework of [FU23] and re-developed the abstract version of the Schnorr protocol and re-derived completeness and proof of knowledge in the context of the latest standard library.

But we also cannot directly apply these proofs to our protocols because of the slightly different interface of the Jasmin code (e.g., state is explicitly passed) but more importantly because the Jasmin code works on low-level representations of the group elements (arrays of bytes) while the existing EasyCrypt implementation works on an abstract type of elements of a cyclic group. This difference is cryptographically relevant – a careless implementation of the abstract specification might introduce insecurities, e.g., by leaking data in the specific choice of representation of a specific group element. We use a principled approach and carry over the properties from the protocol implemented on the high-level of abstraction (such as [FU23]) to the low-level (i.e., protocol implemented in Jasmin). We believe that this approach also provides guidance for similar

[5] One may ask why they should not be written in Jasmin and verified in Jasmin. The reason is that these refer to some hypothetical adversary (more specifically, the are reductions that transform one adversary into another), and there is no reason to assume that this adversary is written specifically in a Jasmin-related assembler.

situations in future work.

Reusability, reproducibility. Throughout this work, we have striven to make our results general and reproducible. While the overall result is specific to the Schnorr protocol, where possible we made individual building blocks independently reusable. The rejection sampling works for arbitrary sets; Barret reduction and modular modularization is independent of the Schnorr protocol; the Montgomery ladder is not specific to modular arithmetic but analyzed for any monoid.

Where this was not possible, we tried to make sure that the overall structure of our results is clean and simple to understand, and tried to explain them in this paper in a way that makes it easy to understand especially the modeling and overall structure of our proofs to enable future work on other protocols that follows our work with as little changes as possible.

Our development is available as a GitHub repository [FOU23].

1.1 Related Work

The Jasmin toolchain was introduced with Coq proofs of the correctness of the compiler in [ABB+17]; it was connected to EasyCrypt in [ABB+20]; the toolchain was extended to cover leakage-freeness guarantees in [BGLP21, SBG+22]. Several cryptographic schemes have been implemented in Jasmin: the ChaCha20 streamcipher, the Poly1305 and Gimli hash function (all in [ABB+20]), the scalar multiplication algorithm for the elliptic curve Curve25519 in [ABB+17], the SHA-3 hash function in [ABRB+19], the Kyber public-key encryption scheme in [ABB+23], and the MPC-in-the-head protocol in [ABC+21]. Of these, most only contain functional correctness proofs. Only [ABC+21] contains a study of security properties, but not of the Jasmin code itself. (See below.) We are not aware of prior work that verifies the *security* of the assembly in EasyCrypt.

Two of these works merit closer attention in the context of our work, [ABC+21] and [ABB+23].

The work [ABC+21] has a similar aim as our work – it analyses a zero knowledge proof system in EasyCrypt, and parts of that protocol are implemented in Jasmin. The protocol is different from the one considered here, it is the MPC-in-the-head (MitH) protocol from [IKOS07]. They prove in EasyCrypt that MitH is a restricted variant of zero-knowledge,[6] and that it has soundness. They also aim at end-to-end verification of the protocol but with a different approach than we do here: They implement the protocol in EasyCrypt and translate the EasyCrypt code to OCaml (using code extraction). Then they re-implement *some* of the operations (namely, the code for addition and multiplication gates) and show that this code is equivalent to the EasyCrypt code and then link it

[6] Restricted in the sense that they only show that there is a simulator that fails with constant probability, while for zero-knowledge, we would need to have a simulator that succeeds with overwhelming probability. They prove a meta-theorem that such an aborting simulator implies the existence of the desired simulator, but as explained in more detail in [FU23], the meta-theorem cannot be applied to the aborting simulator that is constructed due to a lack of generality of the meta-theorem. (It only applies to simulators that are of the specific form that they have just one step, side-effect free sampling from a given distribution.)

together with the OCaml code using a special-purpose C wrapper. Compared to a full end-to-end verified implementation *in Jasmin* (as we do here for Schnorr's protocol), this has some limitations. First, the joining of the autogenerated OCaml code, and the manually written Jasmin code is unverified: While there is an EasyCrypt proof that the Jasmin code is equivalent to the EasyCrypt code in certain sense, there is no principled method that checks that the interfacing is done properly; we trust the unverified implementation of the wrapper do deal with combining a high- and a low-level language in a way that makes their semantics match up. (This is non-trivial since OCaml is a garbage collected language, and the wrapper needs to "understand" the foreign function interface of OCaml and deal with memory allocations etc.) Compared with an approach that verifies everything in one language, this introduces additional points of failure. Second, we lose one of the selling points of Jasmin, the leakage-freeness. No guarantees can be given about the leakage (e.g., though timing) of the OCaml code, and even if the Jasmin code fragments are constant-time, this property is lost when the data is processed by the OCaml code. We also stress that the security proofs are about the EasyCrypt implementation of the overall program – for the Jasmin programs, functional properties are shown. In our present work, we aim to avoid those limitations by implementing all EasyCrypt code in Jasmin, and verifying in EasyCrypt that the security properties hold specifically for the Jasmin code.

The recent independent work [ABB+23] that analyses various Kyber [BDK+18] implementations in Jasmin and analyses their functional correctness (i.e., that they match the abstract specification in EasyCrypt). We stress that they do not cover security properties of Kyber, nor leakage-freeness. (For the latter, our new definitions of leakage-freeness would come in handy since their implementations also are not constant-time.) Their work also contains an implementation of Barrett reduction, and of rejection sampling. Their implementations are very specialized to Kyber: Their Barrett reduction is specifically for 16-bit words and hence does not apply to integers modulo a large prime. 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).[7] In contrast, our Barret reduction is for large integers, too, and our rejection sampling is generic (and provably uniform).[8] Additionally, we obtain leakage-freeness results for both.

[7] 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.

[8] Using our rejection sampling in Kyber would be indistinguishable from the one in the specification but this would probably violate the (not yet published) standard.

2 Preliminaries

2.1 EasyCrypt

EasyCrypt 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 [BDG+13].

This paper was inspired by the formalization of zero-knowledge protocols in EasyCrypt [FU22]. To facilitate reading, in this paper we used the same style of presentation and set of syntactical conventions as [FU22]. To our readers who are not familiar with EasyCrypt also suggest to read a short EasyCrypt introduction in [FU22, Section 2] or in Appendix A. More information on EasyCrypt can be found in the EasyCrypt tutorial [BDG+13].

To readers who are familiar with EasyCrypt we only give a brief overview of our syntactical conventions: we write $\leftarrow$ 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 Workbench

Jasmin is a toolchain for high-assurance and high-speed cryptography [ABB+17]. 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 on a simple example. More specifically, we implement a program `rinv(x)` which with equal probabilities must return `x` or `-x`, where `-x` denotes a binary word with all bits of `x`

being flipped. Below is the straightforward implementation of such a program in Jasmin:

```
inline fn rinv(reg u8 x) → (reg u8) {
  stack u8[1] q;

  q = #randombytes(q);
  if(q[0] < 128){
    x ^= 0xff; // x ⊕ 255
  }
  return x;
}
```

The function `rinv` has one argument which is 8-bit unsigned word allocated in register (denoted by type `reg u8`). The program starts by sampling a random byte with a systemcall `#randombytes`. The systemcall takes a 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 `b[0]`. Hence, with probability `1/2` the value `b[0]` is smaller than `128` and the result of computation is `x ^ 0xff`; otherwise, we return the value `x` unchanged.

To address correctness of `rinv` we can instruct the Jasmin compiler to extract an EasyCrypt model of `rinv` program.[9] This produces a module $\mathtt{M_C}$ with a procedure `rinv`. 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 <$- duniformW8A1;
    return a;
  }
}.

module M_C(SC:Syscall_t) = {
  proc rinv (x:W8.t) : W8.t = {
    var q:W8.t Array1.t;
    q ← witness;
    q <@ SC.randombytes_1 (q);
    if ((q.[0] \ult (W8.of_int 128))) {
      x ← (x `^` (W8.of_int 255));
    } else {

    }
    return x;
  }
}
```

[9] The command `jasminc -ec rinv -oec Rinv.ec Rinv.jazz` tells Jasmin compiler to extract to EasyCrypt the function `rinv` from Jasmin source file `Rinv.jazz` and produce an EasyCrypt source file `Rinv.ec`.

For example, the Jasmin datatype `reg u8` of 8-bit words was translated to the EasyCrypt type `W8.t`. The single-entry 8-bit array `stack u8[1]` was translated to a single-entry 8-bit word `W8.t Array1.t`. EasyCrypt does not recognize a difference between values allocated on stack and in registers, so this information is abstracted away during translation.

Since `rinv` program in Jasmin uses a systemcall `#randombytes` then Jasmin generates a module $\mathtt{M_C}$ which is parameterized by a "provider" of systemcalls `SC`. In our example, the systemcall `#randombytes` is translated to an invocation of $\mathtt{SC.randombytes}_1$ 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 $\mathtt{SC_D}$ with the "default" interpretation of systemcalls. In our work we use the default interpretation which models `#randombytes` as a generator of truly random bytes (denoted by distribution `duniformW8A1`). Alternatively, one could interpret `#randombytes` as an invocation of pseudo-random generator.

If we assume that the translation from Jasmin to EasyCrypt preserves the computational semantics then we can use the EasyCrypt's module $\mathtt{M_C}$ to address the correctness of our Jasmin implementation. More specifically, we use the EasyCrypt's built-in probabilistic Hoare logic to prove the following property:

```
∀ x, Pr[ r ← MC(SCD).rinv(x)@m: r = x ]
       = Pr[ r ← MC(SCD).rinv(x)@m: r = -x ].
```

Hence, we conclude that `rinv` is implemented as desired which means that the outputs `x` and `-x` are equally likely.

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 (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 $\mathtt{M_L}$ has a global variable `leakages` which is used in the EasyCrypt procedures to accumulate information which can get leaked in case of a timing attack. For example, if we compile `rinv` program to EasyCrypt with leakage annotations then the result is as follows:[10]

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

  proc rinv (x:W8.t) : W8.t = {
    var b;
    leakages ← LeakAddr [] :: leakages;
```

[10]The command `jasminc -CT -ec rxor -oec Rinv_ct.ec Rinv.jazz` tells Jasmin compiler to extract to EasyCrypt the function `rxor` from Jasmin source file `Rinv.jazz` and produce an EasyCrypt source file `Rinv_ct.ec`.

```
    b <@ SC.randombytes_1 (b);
    leakages ← LeakCond(b.[0] < W8.of_int 128) :: leakages;
    leakages ← LeakAddr [] :: leakages;
    if ((b.[0] \ult (W8.of_int 128))) {
      leakages ← LeakAddr [] :: leakages;
      x ← (x `^` (W8.of_int 255));
    } else {}
    return x;
  }
}
```

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 $\mathtt{M_L.rinv}$ the bitwse xor operation `(^)` only adds a leakage value `LeakAddr []` to the `leakages` accumulator. This means that the `(^)` operation does not leak any data about its arguments nor about the result of the computation. At the same time, since the execution of `(^)` requires time then this is modelled by adding an empty leakge value `LeakAddr []`.

Also notice that `if`-statements leak the boolean value of the conditional statement. As a result the boolean value `b.[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 `rinv` is not leakage-free since it leaks some data about the actual dataflow of the program execution.

Let us implement a new version of the `rinv` program which gets rid of the problematic `if`-statement:

```
inline fn rinv(reg u8 x) → reg u8 {
  stack u8[1] b;
  reg u8 y;
  b = #randombytes(b);
  b[0] &= 1;
  y = 0xff;
  y &= b[0];
  x ^= y;
  return x;
}
```

In the new version of `rinv` program we convert a random byte `b[0]` to the values `0` or `1` by doing a bitwise "and" operation of `b[0]` with `1` and store the result back in `b[0]`. Next, we multiply the `b[0]` by a value `0xff` after which with equal probabilities `b[0]` must be equal to either `0` (all bits are zero) or `0xff` (all bits are `1`). Next, we do a bitwise "and" of `b[0]` and `y` (at this stage with equal probabilities `y` must be either equal to `0` or remain equal to `0xff`). Finally, the "xor" of `x` with `y` is returned.

We can prove that the new version of `rinv` computes the same probabilistic function as our first attempt. 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 M_L(SC:Syscall_t) = {
  var leakages : leakages_t

  proc rinv (x:W8.t) : W8.t = {
    var b:W8.t Array1.t;
    var y:W8.t;
    b ← witness;
    leakages ← LeakAddr([]) :: leakages;
    b <@ SC.randombytes_1 (b);
    leakages ← LeakAddr([0]) :: leakages;
    b.[0] ← (b.[0] `&` (W8.of_int 1));
    leakages ← LeakAddr([0]) :: leakages;
    leakages ← LeakAddr([]) :: leakages;
    y ← (W8.of_int 255);
    leakages ← LeakAddr([0]) :: leakages;
    y ← (y `&` b.[0]);
    leakages ← LeakAddr([]) :: leakages;
    x ← (x `^` y);
    return (x);
  }
}.
```

Now it must be easy to see that `leakages` accumulator does not contain any data specific to the input or output of the program. Moreover, the same list of leakages is generated on every execution of the `rinv` function (i.e., the resulting $\mathtt{M_L}$`.leakages` is deterministic and not probabilistic). Therefore, we can conclude that the second Jasmin implementation of the `rxor` function is leakage-free.

However, to be able to argue about constant-time property formally we must give a general definition of constant-time programs in Jasmin.

We assume that Jasmin programs have public inputs and secret inputs. The intention of our definition is to guarantee safety against timing attack. In other words, we want to ensure that programs which satisfy our definition of leakage-freeness must not leak any information about their secret inputs and the result of their computation through timing-attacks.

Definition 1 (Leakage-Free Jasmin Programs) *Let* `p` *be a Jasmin program and* $\mathtt{M_L}$ *be the result of extraction of* `p` *to EasyCrypt with leakage annotations. Also, let* `pin` *and* `sin` *be public and secret inputs, respectively. Then we say that* `p` *is a leakage-free program with respect to systemcall provider* `SC` *iff:*

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

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

Let us apply this definition to `rinv` function (assuming the default systemcall provider). We also assume that input `x` is secret (in which case there are no public inputs) then we define function `f` as follows:

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

Here, `f` checks if the list of leakages `l` is well-formed (i.e., equals to a constant list denoted by `rxor_l`) in which case it returns `1`, and `0` otherwise.

By using the basic EasyCrypt reasoning we can prove that the Jasmin program `rinv` with functions `f` as defined above satisfy the definition of being leakage-free according to Definition 1.

For deterministic[11] programs we can give alternative definition of leakage-freeness property which for such programs is equivalent to Definition 1, but in EasyCrypt the latter definition could usually be established almost entirely automatically by using probabilistic relational Hoare logic with combination of simulation tactic `sim`:

Definition 2 (Deterministic Constant-Time Jasmin Programs) *Let* `p` *be a deterministic Jasmin program and* $\mathtt{M_L}$ *be the result of extraction of* `p` *to EasyCrypt with leakage annotations. Also, let* `pin` *be a public input and* $\mathtt{sin}_1$ *with* $\mathtt{sin}_2$ *be secret inputs. Then we say that* `p` *is a constant-time program with respect to systemcall provider* `SC` *iff:*

```
∀ sin1 sin2 pin l m, ML.leakages{m} = []
 ⇒ Pr[ ML(SC).p(pin,sin1)@m: ML.leakages = l ]
   = Pr[ ML(SC).p(pin,sin2)@m: ML.leakages = l ].
```

We also formally prove in EC that Definition 2 implies Definition 1. The proof could be found in supplementary code (lemma `lf_implies_ct` in the file `proof/definition_analysis/Defs.ec`).

3 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. Therefore, 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 $[0 \dots 7]$ 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.

[11] In EasyCrypt we say that the program `p` is deterministic iff there exists a (pure) function which computes the same outputs as `p` for all inputs.

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. Nonetheless we can prove that if the source distribution d has elements which satisfy the criteria then the rejection sampling is always terminating.[12]

In this section, we first start by implementing a generic rejection sampling algorithm in EasyCrypt and proving its properties (termination and correctness). Then 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. Finally, we extract the Jasmin sampling algorithm to EasyCrypt with leakage annotation and prove that it is leakage-free.

3.1 Rejection Sampling in EasyCrypt

We start by implementing a rejection sampling algorithm in EasyCrypt. Our algorithm is parameterized by a lossless distribution `d` of 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;
  }
  // also includes rsample1 which unfolds  while-loop once.
}.
```

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 probabilistic

[12] We considered implementing alternative algorithms which have apriori termination time, however to the best of our knowledge, those only are able to produce approximations of target distributions.

relational Hoare logic (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 ].
```

Observe that if `rsample(P)` terminates then the returned element `x` must satisfy the predicate `P`. Therefore, in the rest we will address only predicates `Q` which define subsets of `P` (denoted by `Impl Q P`). In the next step towards correctness of `rsample` 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: Impl Q P
 ⇒ Pr[ x ← RS.rsample1(P)@m: Q x ]
     = (μ d !P) * Pr[ x ← RS.rsample(P)@m: Q x ] + μ d Q.
```

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

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

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

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

We can also rewrite the right-hand side as `(μ d Q)/(μ d P)` which denotes a condtional 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
   ⇒ is_lossless d
   ⇒ Pr[ x ← RS.rsample(P)@m: P x ] = 1.
```

3.2 Uniform Sampling in Jasmin

In Jasmin we cannot implement a rejection sampling algorithm as generic as our implementation of `RS.rsample(P)` in EasyCrypt (see Sec. 3.1).[13] For the purposes of Schnorr protocol, we implement in Jasmin a function which specializes the predicate `P` to `λ x. x < a` (for a parameter `a`) and uses `#randombytes`

[13] Jasmin does not have function types, so a predicate cannot be passed as an argument to a program. Also, 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.

systemcall as a distribution `d`. In this way, we implement a uniform sampling from an interval `[0...a-1]` for a given parameter `a`.

Another current restriction of Jasmin language is that 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.[14]

Now we describe an implementation of a Jasmin program `bn_rsample(a)` (prefix `bn` stands for big-number) whose input `a` is a `nlimb`-array representing a number from the interval $[0...2^{64 \cdot \texttt{nlimbs}} - 1]$ which is allocated on stack. The program returns the 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.

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

Next we compile `bn_rsample` to EasyCrypt without leakage-annotations to address correctness. This produces a module $\texttt{M}_\texttt{C}$ with the EasyCrypt's version of `bn_rsample` algorithm. The module also includes all functions which were used

[14] In this paper we define `nlimbs := 32`, but our development can be recompiled with any value.

[15] The implementation of `bn_subc` is included into the Jasmin standard library. Moreover, it comes with associated EasyCrypt proofs which show that it is correct and constant-time.

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.

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 extract $\mathtt{M_C.bn\_rsample}$. In particular, we use the EasyCrypt's built-in probabilistic relational Hoare logic to establish that $\mathtt{M_C.bn\_rsample}$ returns a number `y` with the same probability as the procedure `RS.rsample`:

```
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) ← MC(SCD).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]` do denote a value of type `W64xN.t` converted to usigned integer (in EasyCrypt this is done by using function `W64xN.valR`).

As a consequence of `bn_rsample_spec` and `rsample_pmf` we can immediately conclude the correctness of `usample`:

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

3.3 Leakage-Freeness

In the previous section we discussed the correctness of implementation of `bn_rsample` in Jasmin. In this section we address its leakage-freeness. To do that, we compile `bn_rsample` to an EasyCrypt module with leakage annotations. The result is as follows:[16]

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

  proc bn_rsample(a:W64xN.t): (int * W64xN.t) = {
    var q p i;
    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;
```

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

```
      p <@ SC.randombytes_32 (init_array nlimbs 64);

      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);
  }

  // also includes leakage-annotated bn_subc, bn_copy, bn_set0.
}.
```

Recall that in the implementation of `bn_rsample` the counter `i` is a "logical" variable which we will use to derive properties of `bn_rsample`.

The module $\mathtt{M_L}$ also includes leakage-annotated versions of `bn_subc`, `bn_copy`, and `bn_set0` which we skip here for brevity. The [lib] library contains proofs that these auxiliary functions are correct and constant-time.

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

In the first step of our analysis we derive the probability of `bn_rsample` 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) ← M_L(SC_D).bn_rsample(a)@m: c = i ∧ x = y ]
     = (fail_once [a])^(i-1) / t.
```

Here, `(fail_once x)` 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 `x` 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 $\mathtt{M_L.bn\_rsample}$ the contents of $\mathtt{M_L.leakages}$ 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 prefix = [ LeakAddr []; ...] ++ set0_L ++ [...] in
  let suffix = [ LeakAddr []; ...] ++ copy_L ++ [...] in
  let loop j = repeat (j-1) [ LeakAddr []; 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 $\mathtt{M_L.leakages}$ by proving that the probability of $\mathtt{M_L.leakages}$ 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 m:
  Pr[ (_,x) ← ML(SCD).rsample(a)@m: ML.leakages = l ∧ x = y ]
   = Pr[ (i,x) ← ML(SCD).rsample(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 m:
  Pr[ (_,x) ← ML(SCD).rsample(a)@m: ML.leakages = l ∧ x = y ]
   = Pr[ (i,x) ← ML(SCD).rsample(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 m: let t = 2^nlimbs*64 in
                            let i = inv samp_t l in
  Pr[ (_,x) ← ML(SCD).bn_rsample(a)@m: ML.leakages = l ∧ 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 1). 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: ML.leakages{m} = [] ⇒
  let v = Pr[(_,x) ← ML(SCD).rsample(a)@m: ML.leakages = l ∧ x = y ] in
  let w = Pr[(_,x) ← ML(SCD).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., $\mathtt{M_L.leakages = l}$) and therefore we return probability which corresponds to `bn_rsample` running for exactly `i` iterations. In other case (i.e., $\mathtt{i \leq 0}$) 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`).

4 Barrett Reduction

In the Schnorr protocol parties need to multiply elements of a cyclic group. In practice, group multiplication is usually implemented as multiplication followed by modular reduction. In other words, we need to compute many instances of $ab \bmod m$ for a fixed modulus m (where $0 \leq a, b < m$). At the same time, the naive implementation of modular reduction by using number division is slow. In our work we perform modular reductions using a specialized Barrett reduction algorithm [Bar87]. In Barrett reduction we precompute a so-called Barrett factor for every particular modulus n and thereafter the computations of $ab \mod m$ only involve multiplications, subtractions, and shifts (all of which are faster operations than long division of numbers).

In Sec. 4.1 we implement the Barrett reduction in EasyCrypt and prove its correctness. In Sec. 4.2, we develop the same algorithm in Jasmin and prove its correctness by relating it to the "high-level" EasyCrypt implementation from this section.

4.1 Barrett Reduction in EasyCrypt

To reduce $0 \leq x < m^2$ modulo m, the Barrett reduction computes $t = x - \lfloor xr/4^k \rfloor m$, where $k = \lceil \log_2 m \rceil$ and $r = \lfloor 4^k/m \rfloor$ (known as a Barrett factor). Note that Barrett factor depends only on the modulus m. Also, it is important to understand that in the context of high-perfomance implementations, the factor r is not easy to compute since it requires division by m (which can be arbitrary). As a result the Barrett factor is usually precomputed and hardcoded for a given m. It is easy to see that $t \equiv x \mod m$, but it is also true (but not trivially) that $0 \leq t < 2m$. In our work, we prove this fact by mostly following an elegant proof from [Nay19]:

```
op t (x m k : real) : real =
 let k = ceil  (log2 m) in
 let r = floor (4^k/m)  in x - (floor (x*r/4^k))*m.

lemma barrett_bound: ∀ x m k,
  ⇒ 0 ≤ x < m*m
  ⇒ 0 ≤ m < 2^k
  ⇒ 0 ≤ t x m k < 2*m.
```

Hence, $x \mod m$ equals to either t if $t < m$ or $t - m$, otherwise. Notice that the above function `t x n` computes with real numbers – this greatly simplified the proof of lemma `barrett_bound`. Next, we implement the `barrett_reduction` function which receives positive integers `x` and `m` then computes `t' := (ti x m k)` (here, function `ti` implements the same computations on integers as `t` on reals) and outputs `t'` if `t' < m` and `t' - m`, oth-

erwise. To avoid confusion we stress that the function `barrett_reduction` is never executed on its own, but it merely acts as a computational algorithm for which we derive correctness. In the next section, we implement the same Barrett reduction algorithm in Jasmin. The Jasmin implementation is then proved to be correct by relating its computations to the `barrett_reduction` function.

```
op barrett_reduction (x m k : int) : int =
  let t' = ti x m k in (if t' < m then t' else t' - m).

lemma barrett_reduction_correct (x m k : int) :
   0 ≤ x < m*m
   ⇒ 0 < m < 2^k
   ⇒ 0 ≤ k
   ⇒ barrett_reduction x m k = x %% m.
```

The correctness of `barrett_reduction` is derived from `barrett_bound` by relating computations of function `ti` to computations of function `t`.

4.2 Barrett Reduction in Jasmin

The function `bn_breduce(r,x,m)` implements the Barrett reduction algorithm in Jasmin. The input is the Barrett factor `r` (precomputed for the modulus `m`), and the number `x` which we want to reduce modulo `m`. Recall that in Jasmin we cannot implement a function whose input would be an array of a parameter size. As a result, the implementation of the Barrett reduction in Jasmin turns out to be a bit cumbersone since we need to track the size of all intermediate results and then use procedures which work with inputs of exactly that size. For example, the input `x` to the `bn_breduce` function is `2*nlimbs`-long (recall that `x` is a result of multiplication of `a` and `b` where $0 \leq$ `a,b` $<$ `m`). Hence, we cannot multiply `x` and `r` by using the function `bn_muln` from libjbn library since it only multiplies `nlimb`-long inputs. Instead, we are forced to re-implement the multiplication function (we call it `dbn_muln`) for numbers where the first argument is `2*nlimbs`-long and second is `2*nlimbs`-long and the result is `4*nlimbs`-long. The function call `div2(xr,2*nlimbs)` computes `x*r/4`$^{\texttt{nlimbs*64}}$.[17] The function `dcminusP(mm,t)` is a constant-time implementation of `if t < mm then t else (t - mm)` expression. At the end of the computation, we have the value `t` which must be equal to `x %% m` but the variable is `2*nlimbs`-long so as a final step in `bn_breduce` we use `bn_shrink(t)` which discards the `nlimbs` highest words.

```
inline fn bn_breduce(stack u64[2*nlimbs] r x, stack u64[nlimbs] m)
 → (stack u64[nlimbs]){
  stack u64[nlimbs] xrfd r;
  stack u64[2*nlimbs] xrf xrfm t mm;
  stack u64[4*nlimbs] xr;

  xr   = dbn_muln(x,r);        // x*r
  xrf  = div2(xr, 2*nlimbs); // x*r/4^nlimbs*64
  xrfd = bn_shrink(xrf);
```

[17] In this section `a/b` denotes integer division.

```
  xrfm = bn_muln(xrfd,m);       // (x*r/4^nlimbs*64)*m
  t    = dbn_subc(x, xrfm);      // x - (x*r/4^nlimbs*64)*m
  nn   = bn_expand(m);
  t    = dcminusP(mm, t);       // if t < mm then t else (t - mm)
  r    = bn_shrink(t);

  return r;
}
```

Correctness To prove the correctness of `bn_breduce`, we want to show that it computes the same function as `barrett_reduction` from previous section. To do that, we derive correctness properties for all procedures used in the implementation of `bn_breduce` (i.e., `dbn_muln` computes multiplication, `dbn_subc` computes subtraction, etc.). For example, we established the following correctness property of `bn_expand` function which converts an `nlimbs` number to `2*nlimbs` number without changing its value:

```
lemma bn_expand_correct m a:
 Pr[ out ← MC.bn_expand(a)@m: [out] = [a] ] = 1.
```

After establishing correctness properties for all procedures used in `bn_breduce` we use the probabilistic Hoare logic to relate `bn_breduce` to `barrett_reduction` function. At the end of it, the correctness of `bn_breduce` follows from `barrett_reduction_correct` lemma:

```
op barrett_factor(m : int) = 4^nlimbs*64/m.

lemma bn_breduce_correct n r x m:
 [r] = barrett_factor [m]
 ⇒  0  < [m]
 ⇒ [x] < [m]*[m]
 ⇒ Pr[ out ← MC.bn_breduce(r,x,m)@n: [out] = [x] %% [m] ] = 1.
```

Constant-Time Property The function `bn_breduce` itself and all functions which are used in its implementation are deterministic. Therefore, we can use 2 to establish the constant-time property. Indeed, by using the EasyCrypt's built-in probabilistic relational Hoare logic the proof is simple:

```
lemma bn_breduce_ct m r1 r2 x1 x2 n1 n2 l:
 ⇒ Pr[ ML.bn_breduce(r1,x1,n1)@m: ML.leakages = l ]
    = Pr[ ML.bn_breduce(r2,x2,n2)@m: ML.leakages = l ].
proof. byequiv. progress. sim. auto. qed.
```

4.3 Modular Multiplication

We now implement modular multiplication as "big" multiplication followed by modular reduction:

```
inline fn bn_mulm(stack u64[2*nlimbs] r, stack u64[nlimbs] m a b)
 → stack u64[nlimbs]{
  stack u64[2*nlimbs] c;
  c = bn_muln(a,b);
  a = bn_breduce(r,c,m);
```

```
  return a;
}
```

The correctness of `bn_mulm` is a direct consequence of correctness of "big" multiplciation `bn_muln` and modular reduction `bn_breduce`.

```
lemma bn_mulm_correct_pr n a b m r:
     [a] < [m]
  ⇒ [b] < [m]
  ⇒ [r] = barrett_factor [m]
  ⇒ Pr[ out ← M_L.bn_mulm(r,m,a,b)@n: [out] = [a]*[b] %% [m] ] = 1.
```

Also, since `bn_muln` and `bn_breduce` are deterministic and constant-time therefore constant-time property also carries over to `bn_mulm`:

```
lemma bn_mulm_ct m r1 r2 a1 a2 b1 b2 p1 p2 l:
  Pr[ M_L.bn_mulm(r1,p1,a1,b1)@m: M_L.leakages = l ]
    = Pr[ M_L.bn_mulm(r2,p2,a2,b2)@m: M_L.leakages = l ].
```

5 Montgomery Ladder

In previous section we described the implementation of Barrett algorithm which is used to efficiently compute modular reduction. Using Barrett reduction we derived modular multiplication as a number multiplication followed by modular reduction.

In this section our goal is to use modular multiplication to implement efficient, correct, and constant-time modular exponentiation. The reader might already guess that if we naively implement x^n as $n-1$ multiplications then the resulting program will not be constant-time. For this reason, we implement the Montgomery ladder [Mon87] which is a specialized algorithm which computes exponentiation in constant-time.

In what follows we implement a "high-level" Montgomery ladder algorithm in EasyCrypt and discuss its correctness. We also comment on the Jasmin implementation of Montgomery ladder and present its properties.

5.1 Abstract and Modular Exponentiation

We implement the Montgomery ladder algorithm which computes $x^n = x \cdot x \cdot \ldots \cdot x$ by utilizing the "square-and-multiply" technique. The algorithm is parametric in the "multiplication" operation as long as it forms a monoid. In our EasyCrypt formalization we develop Montgomery ladder algorithm which is parameterized by a monoid. Later we specialize this implementation to the modular multiplication which results in algorithm which computes modular exponentiation.

To compute x^n the algorithm iterates over the binary representation of the power n, starting from the most significant bit (in our work, we use the convention that the most significant bit is the rightmost one). We assume that n is given as a binary string (i.e., list of booleans) of a fixed length L. The variable i is initialized with $L-1$. The main computations are performed in the loop which

at each iteration decreases i by 1 and the loop runs while $0 < i$. Over the course of its computation the Montgomery ladder algorithm maintains the following loop-invariant: $x_1 = x^{n_{>i}}$ and $x_2 = x^{n_{>i+1}}$, where $n_{>i}$ denotes the number n with its i lowest bits dropped. Hence, if the invariant is preserved then after loop terminates (i.e., $i = 0$) we can return x_1 as a result of the computation.

Let us address the invariant preservation. At the beginning ($i = |n|$), we have $n_{>i} = 0$, so the invariant holds initially if we set $x_1 = e$ and $x_2 = x$. We assume that the invariant holds for i and show it for $i - 1$. If the $i - 1$-bit is 0 then to maintain the invariant we must simulateneously update variables x_1 and x_2 with values $x_1 \cdot x_1$ and $x_1 \cdot x_2$, respectively. Indeed, $n_{>i-1} = 2n_{>i}$, hence, $x_1 \cdot x_1 = x^{n_{>i}} \cdot x^{n_{>i}} = x^{2n_{>i}}$ and $x_1 \cdot x_2 = x^{n_{>i}} \cdot x^{n_{>i+1}} = x^{2n_{>i+1}}$ as desired. In the other case, when the $(i-1)$st most significant bit is 1 then $n_{>i-1} = 2n_{>i} + 1$ and we must simultaneously update the variables x_1 and x_2 with values $x_1 \cdot x_2$ and $x_2 \cdot x_2$, respectively. It must be easy to verify that the invariant is preserved in this case as well. Due to the symmetry in the above computations we can encode the above with two `swap`s. First, depending on whether the current bit is `0` or `1` we swap the values of $\texttt{x}_1$ and $\texttt{x}_2$: `(x1,x2) ← swap b x1 x2` (here, `b` is the current bit of power `n`). Next, we compute a pair $(\texttt{x}_1\texttt{*x}_1\texttt{, x}_1\texttt{*x}_2)$ and then depending on the bit `b` we assign it to either $(\texttt{x}_1\texttt{,x}_2)$ or $(\texttt{x}_2\texttt{,x}_1)$.

```
// monoid structure
type R.
op ( * ) : R → R → R.
op e : R.

axiom op_assoc (a b c : R) :  (a * b) * c = a * (b * c).
axiom op_id (x : R) :  x * e = x.
axiom op_id' (x : R) : e * x = x.

module ML = {
  proc mladder (x:R, n:bits) : R = {
    var x1, x2, i, b;
    (x1,x2) ← (e,x);
    b ← false;
    i ← size n;
    while (0 < i) {
      i ← i - 1;
      b ← ith_bit n i;
      (x1,x2) ← swap b (x1, x2);
      (x1,x2) ← swap b (x1*x1, x1*x2);
    }
    return x1;
  }
}.
```

Notice that our implementation of Montgomery ladder is as efficient as sequential multiplication in the worst case: our exponentiation uses $2|n|$ multiplications.

Using the invariant which we described above we prove that `ML.mladder(x,n)` correctly computes exponentiation (i.e., iteration of moinoidal operation):

```
lemma mladder_correct (x : R) (n : bits) m:
```

```
Pr[ out ← ML.mladder(x,n)@m: out = exp x (bs2int n) ] = 1.
```

(Here, `exp x n` denotes an `n-1` iteration of monoidal operation, and `bs2int` converts a list of Booleans (type `bits`) to integers.) In Jasmin we implement function `bn_expm(r,m,x,n)` which uses Montgomery ladder to compute modular exponentiation (the argument `r` is the Barrett factor needed by modular multiplication function, see Sec. 4.1). The function `bn_expm` implements the same algorithm as EasyCrypt's procedure `ML.mladder` (see Appendix 2 for the Jasmin code). The function `bn_expm` instantiates "monoidal operation" with modular multiplication `bn_mulm(r,m,x1,x2)` (here, `r` and `m` are fixed parameters). Then we extract Jasmin function `bn_expm` to EasyCrypt and derive its correctness by establishing a relation between `ML.mladder` and $\mathtt{M_C}$`.bn_expm` procedures. After this the correctness of `bn_expm` is a consequence of `mladder_correct` lemma:

```
lemma bn_expm_correct m r m x n:
     [x] < [m]
  ⇒ [r] = barrett_factor [m]
  ⇒ Pr[ out ←MC.bn_expm(r,m,x,n)@m: [out] = [x]^[n] %% [m] ] = 1.
```

The function `bn_expm` can automatically be proved constant-time since it is deterministic, it does not use conditionals, and its implementation relies only on the constant-time functions (such as `swap`, `ith_bit`, and `bn_mulm`).

```
lemma bn_expm_ct m r1 r2 x1 x2 n1 n2 m1 m2 l:
  Pr[ ML.bn_expm(r1,m1,x1,n1)@m: ML.leakages = l ]
   = Pr[ ML.bn_expm(r2,m2,x2,n2)@m: ML.leakages = l ].
proof. byequiv. progress. inline*. sim. auto. qed.
```

The proof here does not use the lemmas that `swap`, `ith_bit`, and `bn_mulm` are constant-time. Instead, `sim` tactic is able to rederive this facts automatically.

6 Schnorr Protocol

The Schnorr protocol is defined for a cyclic group G_q of order q with generator g. The language of the Schnorr protocol consists of all group elements. In the Schnorr protocol the prover tries to convince a verifier that it knows a discrete logarithm of the statement. In other words, if $s \in G_q$ is a statement then a corresponding witness is an element w so that $s = g^w$. The group G_q and the generator g are public parameters. The prover interacts with the verifier as follows:

- The prover chooses a $r \in \mathbb{Z}_q$ uniformly at random and sends $z := g^r$ as the commitment.
- The verifier replies with a challenge $c \in \mathbb{Z}_q$ chosen uniformly at random.
- The prover responds with $t = r + cw$.
- The verifier accepts if $g^t = zs^c$.

The Schnorr protocol is known to have completeness and proof-of-knowledge. In this work we skip the formal definitions of these properties and only provide the intuitive description of these standard properties of ZK protocols.

- Completeness ensures the correct operation of the protocol if both prover and verifier follow the protocol honestly (in other words, exactly as prescribed above).
- Proof-of-knowledge guarantees that any prover that successfully convinces the honest verifier actually knows a witness (and not only abstractly that it exists).

In [FU23], the authors develop a zero-knowledge framework in EasyCrypt and then use it to prove that Schnorr protocol is correct and secure. Their proofs are done in the fully abstract setting. More specifically, they define commitments and statements as elements of an abstract cyclic group. The exponents (i.e., challenge, response, and secret r) of that group are also elements of an abstract type which form a finite field of prime order. As mentioned earlier we adjusted their formalization of Schnorr protocol to make it compatible with the latest edition of EasyCrypt standard library (in the following we will refer to this formalization as "high-level" Schnorr protocol).

The caveat of the "high-level" implementation of Schnorr protocol is that the prover has state (this is forced by the interface requirements of the framework [FU23]). More specifically, the commitment procedure receives a pair of statement and a witness which are not needed to produce commitment but these values are stored in the prover's state for later computations. Unfortunately, Jasmin does not have convenient infrastructure to work with global variables. As a result we decided to implement the stateless versions of prover and verifer. In this implementation the commitment procedure does not take any arguments, but returns a commitment `g^r` (for a randomly sampled `r`) paired with the "secret" exponent `r`. We expect that the handling and dispatching of the secret `r` is handled outside of Jasmin by a program which compiles together the phases of the prover. Also, this implementation is done on the "middle-level" of abstraction as we refine datatypes to be closer to the final "low-level" implementation in Jasmin. More specifically, in our implementation below the types `commitment` and `statement` are defined as synonyms for the abstract type `zmod` which in EasyCrypt denotes a type of elements of an finite field given by integers modulo `p`, where `p` is a prime number. The types `witness`, `secret`, and `response` are synonyms for the type of unbounded integers `int`. Also, our implementation is parameterized by a generator `g` and prime number `q`, so that `g^q = one` (i.e., `g` induces subgroup of prime order `q`).

In EasyCrypt we implement a module `SchnorrProver` with procedures `commitment` and `response` which correspond to the first and third messages of Schnorr protocol. The module `SchnorrVerifier` has procedure `challenge` which computes the second message of the Schnorr protocol and procedure `verify` which decides if to accept the message exchange.

```
module SchnorrProver = {
```

```
  proc commitment(): commitment × secret = {
    var r;
    r ←$ [0..q-1];
    return (g^r, r);
  }
  proc response(w: witness, r: secret,
                c: challenge): response = {
    return r + c * w;
  }
}.

module SchnorrVerifier = {
  proc challenge(): challenge = {
    var c;
    c ←$ [0..q-1];
    return c;
  }
  proc verify(s: statement, z: commitment,
              c: challenge, t: response): bool = {
    var v, v';
    v  ← z * s^c;
    v' ← g^t;
    return (v = v') ∧ s^q = one;
  }
}.
```

With little effort we carried over the proofs of completeness and proof-of-knowledge from "high-level" to "middle-level" Schnorr protocol. At the same time, in this process we needed to address details which are absent on the "high-level". For example, notice that in our "middle-level" Schnorr protocol implementation the `verify` procedure also checks that `s^q = one`. This is needed to guarantee that statement `s` belongs to the subgroup generated by `g`. In the "high-level" implementation such checks are not needed because statements and commitments are definitionally elements of the cyclic group induced by the generator.

In the next sections, our goal is to implement "low-level" Schnorr protocol in Jasmin and prove its correctness and proof-of-knowledge by carrying over these properties from the "middle-level" Schnorr protocol which we defined in this section. Also at the "low-level" we address the leakage-freeness.

6.1 Schnorr in Jasmin

In the following we give "low-level" Jasmin implementation of the main procedures of the prover of the Schnorr protocol. For brevity we skip the detailed description of the Jasmin implementation of the verifier which can be found in the file `src/schnorr_protocol.jazz` of our supplementary code.

In the code below the functions `bn_set_g`, `bn_set_p`, `bn_set_q`, `bn_set_pb` and `bn_set_pq` return (hard-coded) values of generator `g`, group order `p`, exponent order `q`, Barrett parameter for `p`, and Barrett parameter for `q`, respectively (see Sec. 6.3).

```
inline fn commitment() → (stack u64[nlimbs], stack u64[nlimbs]){
  stack u64[nlimbs]  secret_power commitment
                     group_generator group_order exp_order;
  stack u64[2*nlimbs] barrett_parameter;

  exp_order         = bn_set_q(exp_order);
  group_order       = bn_set_p(group_order);
  group_generator   = bn_set_g(group_generator);
  barrett_parameter = bn_set_bp(barrett_parameter);

  _, secret_power = bn_rsample(exp_order);
  commitment = bn_expm(barrett_parameter, group_order,
                       group_generator, secret_power);
  return (commitment, secret_power);
}


inline fn response(stack u64[nlimbs] witness
                                     secret_power challenge)
 → (stack u64[nlimbs]){
  stack u64[2*nlimbs] exp_barrett;
  stack u64[nlimbs]  exp_order response product;

  exp_order   = bn_set_q(exp_order); // [exp_order] = q
  exp_barrett = bn_set_bq(exp_barrett);

  challenge    = bn_breduce_small(exp_barrett,
                                  challenge, exp_order);
  secret_power = bn_breduce_small(exp_barrett,
                                  secret_power, exp_order);
  witness      = bn_breduce_small(exp_barrett,
                                  witness, exp_order);

  product  = bn_mulm(exp_barrett, exp_order, challenge, witness);
  response = bn_addm(exp_order, secret_power, product);

  return response;
}
```

Notice that in the `response` procedure we are using the arguments to compute a response (i.e., exponent). Therefore, to normalize these arguments for multiplication and addition functions we first reduce them modulo exponent order (i.e., modulo prime `q`). The function `bn_breduce_small` implements Barrett reduction algorithm for `nlimbs`-sized words (in Sec. 4.2 we developed procedure `bn_breduce` which reduced values of `2*nlimbs`-sized, since these values were results of the "big" multiplication).

Leakage-Freeness. We also prove that procedures `verify` and `response` are constant-time and `commitment` and `challenge` procedures are leakage-free. Indeed, `response` and `verify` are deterministic, so by using the EasyCrypt's `sim` tactic we almost entirely automatically conclude that these procedures are constant-time with respect to Definition 2. The procedures `challenge` and `commitment` are probabilistic, so we prove that both are leakage-free with respect to Definition 1 by using similar approach as we carried

out for the rejection sampling program `bn_rsample` (see Sec. 3).[18]

6.2 Properties for Schnorr in Jasmin

Recall that in the beginning of this section we presented the "middle-level" Schnorr protocol by implementing EasyCrypt's modules `SchnorrProver` and `SchnorrVerifier`. Moreover, we explained that for these modules we have derived completeness and proof-of-knowledge properties by carrying them over from the "high-level" Schnorr protocol. Both versions of the Schnorr protocol are defined in terms of "analytical types" which allows us to carry out simpler proofs which do not need to worry about low-level details like overflow, representation of group elements, exponents as bitstrings, etc.

At the same time the low-level representation is important if we want to execute the protocol on the real-world computers. For example, a verifier needs to know how many bits to read from the network when it receives a commitment from a prover. Also, the honest prover wants assurance that when it computes commitment then no information is leaked from the side-channel.

Another important aspect is that low-level representation of data in cryptographic protocols is important for "high-level" security properties like completeness and proof-of-knowledge. For example, `SchnorrVerifier` and `SchnorrProver` compute with group elements which have unique representation (property of elements of type `zmod`) which greatly simplifies the proofs. At the same time, in real-world where group elements are represented as bitstrings (e.g., as elements of type `W64xN.t`) the same group element can have multiple representations (e.g., reduced or not reduced modulo group order). Therefore, to prove that real-world protocol is secure we also needs to prove that it handles different representation of data correctly.

We start by extracting Jasmin code to EasyCrypt. The result of the extraction are EasyCrypt modules `JProver` and `JVerifier` (here, we assume that both are already applied to default systemcall provider).

In our approach, to establish completeness and proof-of-knowledge for the low-level Jasmin implementation we first prove a relation between "middle-level" procedures of `SchnorrProver`, `SchnorrVerifier` and "low-level" procedures of `JProver` and `JVerifier`. More speecifically, we show that for any arguments (normalized or not) procedures compute the same elements of the respective groups. For example, we use probabilistic relational Hoare logic (pRHL) to establish the following equivalence for the verify procedures:

```
lemma verify_eq: equiv [ SchnorrVerifier.verify ~ JVerifier.verify:
                           asint s{1} = [statement{2}]  %% p
                         ∧ asint z{1} = [commitment{2}] %% p
                         ∧ c{1} %% q  = [challenge{2}]  %% q
                         ∧ t{1} %% q  = [response{2}]   %% q
```

[18] In the future we plan to prove that the composition of constant-time procedures with leakage-free procedures is leakage-free. This result would simplify the proof that `commitment` is leakage-free.

```
                          => (res{1} = true) = (res{2} = W64.one)].
```

(Here, `asint` is an injection from `zmod` to integers.) Intuitively, the lemma states that if inputs of verify procedures are equal as elements of the respective groups (represented as integers) then the `SchnorrVerifier` accepts (i.e., returns `true`) iff `JVerifier` returns binary constant `W64.one`.

We also prove similar results for the remaining three procedures (i.e., `commitment_eq`, `challenge_eq`, and `response_eq`). Then we use these results to prove the completeness and proof-of-knowledge for `JProver` and `JVerifier` from completeness and proof-of-knowledge for `SchnorrProver` and `SchnorrVerifier`.

Let us illustrate our approach on the completeness property. The completeness for Jasmin implementation is defined as the following "game":

```
module CompletenessJ = {
  proc main(s:W64xN.t, w:W64xN.t) = {
    var z, c, r,t,v;
    (z,r) <@ JProver.commitment();
       c  <@ JVerifier.challenge();
       t  <@ JProver.response(w,r,c);
       v  <@ JVerifier.verify(s,z,c,t);
    return v ≠ W64.zero;
  }
}.
```

This module encodes a message exchange between honest prover and honest verifier. The goal is to prove that in `CompletenessJ` the verifier always accepts.

In the similiar way (*mutatis mutandis*) we defined completeness module `CompletenessG` for "middle-level" Schnorr protocol and then we derived the following completeness lemma from "high-level" Schnorr protocol:

```
lemma completenessG (s: zmod) (w: int)@m: (inzmod g)^w = s
 ⇒ Pr[ r ← CompletenessG.main(s, w)@m : r ] = 1.
```

(Here, `inzmod` is an embedding of integers into finite field `zmod`.) Next, we prove equality of success probabilities of `CompletenessJ` and `CompletenessG` games:

```
lemma completeness_eq m (s w: W64xN.t): [g]^[w] %% p = [s] %% p
  ⇒ Pr[ r ← CompletenessJ.main(s,w)@m : r ]
     = Pr[ r ← CompletenessG.main(inzmod [s], [w])@m : r ].
```

Here, we emphasize that the proof of `completeness_eq` is only 12 lines of code where the main step is the sequential use of `verify_eq`, `response_eq`, `challenge_eq`, and `commitment_eq` lemmas.

The lemma `completenessG` together with `completeness_eq` immediately imply that "low-level" Schnorr protocol as defined by `JProver` and `JVerifier` has completeness.

Using the same approach we also establish proof-of-knowledge for "low-level" Schnorr protocol in a matter of few lines of EasyCrypt code.

6.3 Instance of Schnorr Protocol

On the EasyCrypt side our results depend on the following parameters and constraints:

```
op g, p, q, bp, bq : int

axiom q_prime : prime q.
axiom p_prime : prime p.
axiom bp_correct : bp = barrett_factor p.
axiom bq_correct : bq = barrett_factor q.
axiom g_correct : g^q %% p = 1.
axiom g_less_p  : 1 ≤ g < p.
axiom q_less_p  : q < p.
axiom q_val_prop1 (x : W64xN.t) : [x] < q*q.
axiom p_less_modulusR : p < 2^nlimbs*64
```

Here, `g` is a generator, `bp` and `bq` are Barrett parameters for primes `p` and `q`, respectively. The lemma `g_correct` makes sure that `g` is a generator of subgroup of order `q`. The lemma `q_val_prop1` establishes that any value in the interval `[0..2^nlimbs*64 - 1]` must be smaller that `q*q`. This property is needed to make sure that we can use Barrett reduction (see Sec. 4) to normalize any element of type `W64xN.t` (on the practical side, this property tells that binary representation of `q` as `W64xN.t` cannot have half of its highest bits equal to zero). The lemma `p_less_modulusR` tells that `p` must fit into `W64xN.t` datatype.

To run and test our development we instantiate these parameters. In particular we set `nlimbs := 32`, so `W64xN.t` is a type which corresponds to 2048-bit values. So, we choose `p` as a 2048-bit safe prime from RFC 3526 [KK19]. The prime `q` in this case is also 2048-bit value which is equal to `(p-1)/2` and generator `g` is equal to 2. In EasyCrypt we prove that these values satisfy all the constraints listed above except of primality of `p` and `q`. Unfortunately, we do not know how to efficiently prove primality of big numbers in EasyCrypt.

Also, the above parameters must be embedded into Jasmin implementation of the Schnorr protocol. Since the numbers `p`, `q`, `bp`, and `bq` are large then in Jasmin they must be encoded as multilimb arrays. To make sure that we do not introduce accidental mistakes when encoding these values we provide a Python script which given primes `p` and `q` produces a file `constants.jazz` with Jasmin implementtion of functions `bn_set_g`, `bn_set_p`, `bn_set_p`, `bn_set_bp`, `bn_set_bq` which are used to return the respective values. To provide full guarantees the script generates the following lemmas and their respective proofs:

```
lemma bn_set_p_correct: phoare[ Mc.bn_set_p:
                                  true => [res] = p ] = 1.
lemma bn_set_q_correct: phoare[ Mc.bn_set_q:
                                  true => [res] = q ] = 1.
lemma bn_set_g_correct: phoare[ Mc.bn_set_g:
                                  true => [res] = g ] = 1.
lemma bn_set_bp_correct: phoare[ Mc.bn_set_bp:
                                  true => [res] = bp ] = 1.
```

```
lemma bn_set_bq_correct: phoare[ MC.bn_set_bq:
                                  true ==> [res] = bq ] = 1.
```

Unfortunately, at the moment Jasmin does not have primitives which would allow us to implement the exchange of messages between parties (i.e., sending messages over the network). As a result, the final implementation of prover and verifier must be done outside of Jasmin. In our development, the final versions of prover and verifier are implemented in C. The role of the C wrapper is to link the protocol procedures (which were previously compiled by Jasmin) and handle a dispatching of messages.

Nontheless, after setting the parameters and discharging the above proof obligations we expect to have guarantees that when Jasmin implementation of Schnorr protocol is compiled to assembly and linked correctly together by the C wrapper then the resulting protocol shall be complete, proof-of-knowledge, and leakage-free.[19]

We give detailed instructions on how to compile and run the protocol in the file `src/README.md` of the accompanying code.

References

[ABB+17] 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.

[ABB+20] 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.

[ABB+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, et al. Formally verifying Kyber episode IV: Implementation correctness. *Cryptology ePrint Archive, Paper 2023/215*, 2023. `https://eprint.iacr.org/2023/215`.

[ABC+21] 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

[19] This claim should be taken with a grain of salt as it only holds modulo large trusted code base which assumes that EasyCrypt is sound, Jasmin compilation and extraction mechanism is correct, system calls for randomness generation return random bytes, etc.

security proofs and implementations of mpc-in-the-head. Cryptology ePrint Archive, Paper 2021/1149, 2021. https://eprint.iacr.org/2021/1149.

[ABRB+19] 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.

[Bar87] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[BDG+13] 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.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BGHB11] 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.

[BGLP21] Gilles Barthe, Benjamin Gregoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. Cryptology ePrint Archive, Paper 2021/650, 2021. https://eprint.iacr.org/2021/650.

[FOU23] Denis Firsov, Tiago Oliveira, and Dominique Unruh. https://github.com/dfirsov/jasmin-zk, 2023. Accessed: 2023-05-24.

[FU22] 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.

[FU23] Denis Firsov and Dominique Unruh. Zero-knowledge in easycrypt. In *Proceedings of the 36th IEEE Computer Security Foundations Symposium (to appear)*, 2023.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[KK19] Mika Kojo and Tero Kivinen. More Modular Exponential Diffie-Hellman groups for Internet Key Exchange. `https://datatracker.ietf.org/doc/rfc3526`, 2019. Accessed: 2023-05-24.

[KSU13] Lee Klingler, Rainer Steinwandt, and Dominique Unruh. On using probabilistic turing machines to model participants in cryptographic protocols. *Theoretical Computer Science*, 501:49–51, 2013.

[lib] BigNums library for Jasmin. `https://github.com/formosa-crypto/libjbn`. Accessed: 2023-05-10.

[Mon87] Peter Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[Nay19] Nayuki. Barrett reduction algorithm. `https://www.nayuki.io/page/barrett-reduction-algorithm`, 2019. Accessed: 2023-05-12.

[SBG+22] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. Cryptology ePrint Archive, Paper 2022/630, 2022. `https://eprint.iacr.org/2022/630`.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology—CRYPTO'89 Proceedings 9*, pages 239–252. Springer, 1990.

A EasyCrypt Basics

In the following we comment on the main constructs of EasyCrypt which include types, operators, lemmas, axioms, theories, module types, and modules. More information on EasyCrypt can be found in the EasyCrypt tutorial [BDG+13].

Types and Operators. EasyCrypt has built-in and user defined types. The examples of built-in types are `bool`, `int`, `real`, and `unit`. Also, every type `t` is associated with a type `t distr` of its discrete (sub)probability distributions. A discrete (sub)probability distribution over a type is defined by its probability mass function, i.e. by a non-negative function from `t` to `real`. Also, EasyCrypt allows users to define recursive datatypes and functions based on a polymorphic typed lambda calculus. On the top-level, pure functions can be defined using the `op` keyword. For example, we can define the negation on booleans as follows:

```
op not (b: bool): bool =
  if b then false else true.
```

In addition, the standard library of EasyCrypt includes the implementation and properties of lists, arrays, finite sets, maps, probability distributions, etc.

Ambient Logic. EasyCrypt has built-in logics which are specialized for reasoning about programs (such as Hoare logic). Furthermore it implements an ambient logic which is a higher-order classical logic for proving mathematical facts and connecting judgements from the other logics. For example, we can use ambient logic to prove that double negation is identity. In lemmas and axioms we will use symbols $\forall$ and $\exists$ instead of official keywords `forall` and `exists`, respectively.

```
lemma notnot: ∀ (b: bool), not (not b) = b.
proof. progress. smt. qed.
```

In EasyCrypt, proofs consist of series of tactic applications which either discharge the proof obligation or transform it into new subgoal(s).

Theories. In EasyCrypt, theories can be used to group together related definitions. Theories can have parameters in the form of declared but undefined operators, types, and axioms. For example:

```
theory MonoidTheory.
  % parameters
  type M.

  op f: M → M → M.
  op e: M.

  axiom assoc: ∀ a b c, f (f a b) c = f a (f b c).
  axiom elaws: ∀ a, f a e = a ∧ f e a = a.

  % more useful results and definitions...
end MonoidTheory.
```

Later, the theory can be "cloned" and the operators and types instantiated with concrete values for which the axioms are provable. This enables modular design of theories.

Modules. In EasyCrypt, cryptographic games are modelled as modules, which consist of procedures written in a simple imperative language. Modules may be parameterized by abstract modules. Modules can be stateful, having global variables. The global variables of a module contains the variables declared in the module and any variables its procedures can access (directly or indirectly).

For example, we can implement a module `BitSampler` which has one procedure and one global variable `log`. The procedure `run` samples a uniform Boolean `b`, adds the result to the log, and returns `b` as the result of the call:

```
module BitSampler = {
  var log: bool list
  proc run() = {
    var b: bool;
    b ←$ duniform [false; true];
```

```
    log ← b :: log;
    return b;
  }}.
```

Note that `BitSampler` does not initialize its `log` variable. In this case, the contents of this variable will depend on the initial memory. In EasyCrypt, the whole memory (state) of a program is referred to by `&m` (or `&n` etc.). If `A` is a module then we can refer to the tuple of all global variables of the module `A` in `&m` as `(glob A){m}`. The type of all global variables of `A` (i.e., the type of `(glob A){m}`) is denoted by `glob A`. For example, `glob BitSampler` equals to `bool list`, and `(glob Bitsampler){m}` is the same as `log{m}` which is the value of `log` variable in memory `&m`.

For readability, we will use syntax $\mathcal{G}_A$ for the type `glob A`. Memories `&m` will be typed in bold without the `&` (i.e., $\mathbf{m}$ for `&m`), and $\mathcal{G}_A^{\mathbf{m}}$ will denote the EasyCrypt value `(glob A){m}`.

Module Types. In EasyCrypt, module types specify the types of the procedures in a module, but say nothing about the global variables of the module.

For example, `BitSampler` can be typed as `Runnable`:

```
module type Runnable = {
  proc run(): bool
}.
```

Probability Expressions. EasyCrypt has built-in `Pr`-constructs which are used to express the probabilities of events in program executions. The general form of `Pr`-expression is as follows: `Pr[program @ initial memory: event]`. For example, the expression `Pr[r ← BitSampler.run() @m: P r]` denotes the probability that the return value `r` of procedure `run` of module `BitSampler` given the initial memory `m` satisfies the predicate `P`.

In EasyCrypt, the program in `Pr`-notation can only be a single procedure call. To simplify the presentation, we relax this restriction and allow us to write multiple statements. In the actual EasyCrypt code the same can be expressed by defining module wrappers with a procedure that contains those statements.

To give an example, we can prove that for any adversary `A`, the success probability of guessing the output of a `BitSampler` is exactly $\frac{1}{2}$. In the following we reuse the `Runnable` module type to universally quantify over adversaries. In EasyCrypt, the notation `M <: T` indicates that the module `M` satisfies the module type `T`.

```
lemma example: ∀ (A <: Runnable{-BitSampler}) m,
  Pr[b₁ ← BitSampler.run();
     b₂ ← A.run() @m: b₁ = b₂] = 1/2.
```

It is important to understand that the module type `Runnable` also includes adversaries (i.e., modules) that read from and/or write to `BitSampler`'s log (e.g., `BitSampler` itself). To exclude such "cheating" adversaries, EasyCrypt allows us to write `Runnable{-BitSampler}` to denote the subset of adversaries whose global variables are disjoint from those of `BitSampler`.

B Montgomery Ladder in Jasmin

```
inline fn bn_expm(stack u64[dnlimbs] r, stack u64[nlimbs] m x n)
 → (stack u64[nlimbs])
{
  reg u64 i b;
  stack u64[nlimbs] x1 x2 x11;

  x1 = bn_set1(x1);
  x2 = bn_copy(x);
  x11 = bn_copy(x1);
  i = nlimbs * 64;
  b = 0;
  while(i > 0){
    i = i - 1;
    b = ith_bit(n,i);
    (x1,x2) = swapr(x1,x2,b);
    x11 = bn_copy(x1);
    x1 = bn_mulm(r,m,x1,x1);
    x2 = bn_mulm(r,m,x11,x2);
    (x1,x2) = swapr(x1,x2,b);
  }
  return x1;
}
```