Oskar Goldhahn

Machine checked proofs for MLWE based encryption using EasyCrypt

Masteroppgave i Master i Matematiske Fag
Veileder: Kristian Gjøsteen
Juni 2023

Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for matematiske fag

Oskar Goldhahn

Machine checked proofs for MLWE based encryption using EasyCrypt

Masteroppgave i Master i Matematiske Fag
Veileder: Kristian Gjøsteen
Juni 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for matematiske fag

Contents

Acknowledgements

Morten Solberg for his guidance in learning and working with EasyCrypt and for helping with an initial outline of this work. Christian Doczkal for his contributions and help in getting the matrix library merged into EasyCrypt. My advisor, Kristian Gjøsteen, for our discussions and his guidance. The EasyCrypt developers for their help with the more tricky parts of EasyCrypt and for acknowledging my issues. Charlotte Mylog for her contributions to the matrix library. My family and friends for their encouragement.

1 Introduction

With the looming threat of quantum computing we need public key cryptography that does not rely on the discrete log problem and other problems that can be solved efficiently on a quantum computer. One approach is to base the cryptography on lattice problems. We define a cryptosystem and prove it secure under lattice based assumptions. For additional assurance we do so using a tool called EasyCrypt, which lets the computer check the validity of your proof. To make this possible we also develop a new matrix library for EasyCrypt that can support dynamically sized matrices.

1.1 Provable Security

Before provable security the security of cryptography relied solely on analysis, the practice of trying to break a cryptographic system. If a system could withstand many attempts at breaking it over a long period of time it was deemed usable. There are multiple problems with this approach. Small changes to a complex system can render it entirely insecure and would thus require new analysis. It is possible for weak parts of the system to be neglected in analysis, but stumbled upon by adversaries.

In an attempt to address these issues much of modern cryptography uses proofs for assurance in addition to analysis. The ideal would be to have a proof that breaking the system is hard for some suitable definition of hard, like there not existing a polynomial time algorithm that breaks the system with significant probability. In cryptography 'significant probability' means probability bounded below by the inverse of some polynomial. Obtaining such a proof would mean solving P = NP, so this standard is too high. The next best thing would be to prove that breaking the cryptosystem implies solving an NP-complete problem in polynomial time, but this has also turned out to be an impractical standard.

In provable security we instead show that a polynomial time algorithm that breaks the cryptosystem would give a polynomial time algorithm solving a well studied not necessarily NP-complete problem that we conjecture to be unsolvable in polynomial time. The hardness of these well studied problems are assumptions and cryptography employing provable security is always proven secure relative to assumptions. Sometimes these cryptographic assumptions do not hold and attacks are found, but we have the advantage of reducing the surface area of these attacks. In the presence of provable security cryptoanalysts can focus their efforts on this smaller surface and adversaries are less likely to stumble upon parts and interactions of the assumptions that have not been wellanalyzed than they would be to stumble upon issues with large cryptographic systems.

Provable security comes with a few additional caveats. Cryptography is implemented in the real world, so we need to model this. The models may be imperfect, allowing attacks that are not captured by the model. As an example many models for computation, especially outside cryptography, fail to model execution time, possibly making the systems vulnerable to timing attacks. Spectre [15] takes this even further and exploits common CPU optimizations to do timing based attacks even on code that on paper should have constant runtime. Avoiding this involves using more accurate models or at least employing targeted countermeasures against the extra-model attacks that are found. There is also no way to guarantee that the proofs are correct. In this work we use Formal Verification to decrease the chances of and to detect errors in cryptographic proofs.

1.2 Formal Verification

Sometimes published proofs have significant gaps or even errors. Formal verification tries to address this by enabling the automation of checking mathematical proofs, and sometimes even automating finding the proofs.

This has been used to great effect in mathematics in cases where there have been doubts on the correctness of the proofs. Gonthier used Coq to write a formally verified proof for the 4 color theorem [12], which had been contentious because of its use of software to exhaustively check a large number of cases. Peter Scholze also had a proof which he could not fully trust humans to verify the correctness of, so he tasked the community using [22] with formally verifying the proof. The resulting work demonstrated that these tools are mature enough to deal with complex mathematics [11].

In computer science formal verification is used to reason about the correctness of algorithms, programs and even hardware. Hardware Description Languages like SystemVerilog [21] are used by hardware vendors to construct an abstract description of the hardware. The abstract machine can then be proven to have the desired behavior and this behavior can be compared to the behavior of the real hardware to automatically search for bugs. Hardware bugs can be very expensive as exemplified by the pentium bug, which caused early Intel Pentium processors to give the wrong result for certain floating point divisions and cost Intel $475 million in replacements alone [14].

Formal verification makes a lot of sense for cryptography where even a small error can have a major impact. This stands in contrast to many other applications of mathematics where some error is acceptable and physical experiments can be used to test that the errors are within tolerances and that the theoretical model works well in practice. It also helps that cryptographers are generally comfortable with programming, which makes it easier to work with and develop proof assistants. We are also able to lean on a lot of the work done by the computer scientists in modeling and reasoning about computation, though there are some differences between the requirements of cryptographers and most computer scientists.

There are some caveats regarding formal verification. The tools with automatic proof generation struggle with some proofs, while the human guided interactive theorem provers (proof assistants) make the proving process much slower than would be the case with a conventional paper proof. There is ongoing research to improve this situation, for example by using AI to automate more of the proving process [19]. The benefits of formal verification also break down if there are bugs in the tools. To address this some tools try to minimize the part of the tools that need to be trusted. Finally, the automation of proof checking does not mean that we do not have to look at the proof at all. The bare minimum is to verify that the theorem statement in the tool is the one the author has claimed to prove, which also requires confirming that all the definitions used in the statement are correct, though some of this can be simplified by using standard definitions where possible. Nevertheless, understanding the definitions can be significantly easier than understanding the proofs, which simplifies peer review.

2 EasyCrypt

EasyCrypt is an interactive theorem prover for reasoning about imperative probabilistic algorithms for cryptography. It includes a standard library with useful definitions and proofs and a checker for validating definitions and proofs written in the EasyCrypt programming language. There are other tools with similar goals, such as SSProve [2], FCF [18], CryptoVerif [9] and CryptHOL [8]. One of the things that makes EasyCrypt stand out is its integration with Jasmin [4], an assembly-like programming language for efficient high-assurance implementation of cryptographic primitives, and its tight integration with Satisfiability Modulo Theory (SMT) solvers, tools for automatically proving statements involving quantifiers. In practice these can be treated as magic black boxes and applied freely to quickly prove results that would otherwise require a lot of manual work.

We will provide a short overview on the logic and specification language of EasyCrypt and some of the theory behind it. A reader who wishes to more fully understand the smaller details of the EasyCrypt code is referred to the EasyCrypt reference manual [1] which contains a more complete overview of the EasyCrypt language. For the theoretical foundations there is the in-progress book by Gilles Barthe [7]. Understanding the foundations is not necessary to work with EasyCrypt but it is necessary to understand why EasyCrypt works. The definitions in this section are adapted from these two references.

2.1 Type Theory

Type theory is an alternative to set theory and the foundation for many modern programming languages and almost all proof assistants. While we will not give a full introduction we will summarize the motivation for using it in place of set theory and show how some of it works in EasyCrypt.

In set theory everything is a set and membership and equality are the primitive predicates. Every statement about sets can be written using sets, equality, membership and logical symbols and is just as well formed if we replace one set with another. This has some unfortunate consequences. In set theory the statement $0 \in 1$ is well formed, and might be true or false depending on how you define the natural numbers. Similarly, with the most basic representations for the natural and real numbers $0_{\mathbb{N}} \in \mathbb{R}$ is well formed and false. Another illustrating example is functions. In set theory functions are subsets f of $A \times B$ such that for all $b \in B$ there is a unique $a \in A$ such that $(a, b) \in f$. Function evaluation $f(a)$ can be defined as $\bigcup\{b \in B | (a, b) \in f\}$[1]. It is not obvious why this definition works, but the main point is that set theory has no way to place restrictions on what f and a can be. $f(f)$ is a well formed expression, and so is $a(a)$ and $\emptyset(\emptyset)$. While in practice mathematicians know to avoid these nonsensical expressions, this still matters because a mismatch between theoretical model and mental model makes the theory harder to understand for practicioners. More importantly it hampers formal verification because the search space includes all these nonsensical expressions and statements.

Type theory addresses this by making functions a fundamental building block and restricting which formulas are well formed in a way that mirrors the mental model of the working mathematician. In type theory every expression has a type, usually denoted by the greek letters α, β, γ, τ, etc. The notation $0 : \mathbb{N}$ means that 0 is a well formed expression with type $\mathbb{N}$. A function

[1]Stricly speaking this is function evaluation of functions with values in B. A more general definition would get the image or codomain from f itself.

$f : \alpha \to \gamma$ has type $\alpha \to \gamma$, motivating the notation. Unlike the similar looking $0 \in \mathbb{N}$ from set theory $0 : \mathbb{N}$ is not a statement internally in the formal logic. It is an external statement used to find out whether an expression or formula is well formed. Depending on the theory types can also have very different properties than sets, allowing logical reasoning directly in the type theory, but this is not relevant to users of EasyCrypt where types can be viewed as being sets with some extra rules for forming expressions and formulas.

As an example the following inference diagram tells us how to build well formed expressions using function evaluation;

$$\frac{f : \alpha \to \beta \qquad a : \alpha}{f(a) : \beta}$$

The inference diagram says that if we have a well formed function f of type $\alpha \to \beta$ and a well formed element a of type α, then $f(a)$ is a well formed expression of type β.

2.1.1 Inductive datatypes

In EasyCrypt types can be defined by induction. The following definition, which we will explain, shows how one might define the natural numbers:

```
type nat = [O | S of nat].
```

This defines the type `nat` along with constructor `O` of type `nat` and constructor `S` of type `nat -> nat`. The definition only mentions the arguments to the constructors because it is implicit that they return elements in the type being defined. All elements in `nat` can be constructed using one of the constructors in a unique way. This means we can do case analysis, that equality is structural on the chain of constructors used and we can use structural induction.

Functions, called operators in EasyCrypt, on these inductive types can be defined using structural recursion as shown in the following definition of addition:

```
op add (x y: nat): nat =
   with x = O => y
   with x = S x' => add x' (S y).
```

The operator above is named `add`, takes two arguments `x` and `y` of type `nat`, and does the following; if `x` is `O` then we return `y`, otherwise we increment `y`, decrement `x` by taking its preimage under `S` and add them together. The match arm `with arg = pattern => expr` pattern matches `arg`, handling the case where it is of the form of `pattern` and returns the expression `expr` for that case. The pattern can assign the preimage to variables, and the expression can use those variables. Pattern matching should be exhaustive, meaning there should be at least one match arm that matches any given value. If there are multiple the earlier arm is given priority.

Recursive definitions should be well founded. There should always be an argument where the constructors are being unfolded and only the inner value is given as an argument to the recursive call. In this case this is `x`. Well founded recursion guarantees that the function terminates, which is needed for function application to give a well defined value.

We can also build upon prior definitions.

```
op timestwo x = add x x.
```

Here the type system allows us to elide the type annotations on both `x` and the `timestwo` operator, since the types can be inferred from the argument and return type of `add`.

We can then state lemmas about the operators and types we have defined and prove them using built-in proof techniques and existing lemmas in the standard library. In EasyCrypt logical statements are well formed formulas with type `bool`. In the type system quantifiers are maps from boolean valued functions to the booleans and equality is a map from pairs of the same type to the booleans. The latter can be verified by printing the operator, `print (=).`, which outputs `op (=) ['a] : 'a -> 'a -> bool.`[2]. The `'a` notation, with a `'` followed by an identifier, is a type parameter, which we will explain more in depth in Section 2.1.2. This is not to be confused with `a'`, where the `'` is part of the identifier and is used by convention when we want a new object similar to `a` in some way.

```
lemma timestwo0: timestwo 0 = 0.
proof. done. qed.
```

`timestwo0` states that using the `timestwo` operator on `0` yields `0`. The proof amounts to a simple computation, so we can use `done` proof tactic[3], which does simplification and some trivial proof techniques like proving equality using reflexivity. In this case it works because it can use the definition of `timestwo` to simplify the left side to `0` and then use reflexivity to prove `0 = 0`.

[2]This does not work for quantifiers because they are not operators internally in EasyCrypt and use special syntax.

[3]proof steps in proof assistants are commonly called ‘tactics’

2.1.2 Type Constructors

EasyCrypt also permits a restricted form of generalization over types.

The definition of lists looks something like

```
type 'a list = [nil | cons of ('a * 'a list)].
```

A list with members of type `'a` is either empty, `nil`, or a pair containing an element in `'a` and the rest of the list `'a list`. The `*` operator is the product type, which is analogous to the cartesian product of two sets. The definition uses the generic type `'a` not referring to any specific type, which means that we don't have to define lists of booleans and lists of numbers separately. A `nat list` is what you would get by replacing all the `'a`-s in the definition with `nat`-s. `list` is a type constructor that takes a type and outputs the type of lists where the type of the elements are given by the argument.

This construction is fairly limited. We cannot rely on any specific properties of the generic type. It is, after all, generic. Formally `list` is a type constructor, a function mapping types to types. In EasyCrypt such type constructors always make a unique type. One way of phrasing this is that type constructors are injective and there is no overlap in the image of different type constructors. `*` can also be seen as a type constructor with two arguments[4].

We can also use a similar mechanism to define operators and prove lemmas that are generic over the type.

```
op zeromap (x: 'a) = 0.

lemma eq_trans (x y z: 'a): x = y => y = z => x = z.
proof. done. qed.
```

The `(x y z: 'a)` notation is shorthand for universal quantification over the type `'a`. Type parameters to operators can usually be inferred from the context, but when they cannot, such as with `nil`, it is possible to provide it as an argument as follows: `nil<:nat>`.

2.1.3 Axiomatic Definitions

Another way of constructing types is using axioms. It is possible to define a type without mentioning any details; `type unit.`. We can then specify the details by

[4] It's actually a family of operators with different arities, since `nat * nat * nat`, `nat * (nat * nat)` and `(nat * nat) * nat` are all different types

using axioms about the type. For the unit type we can add an axiom that it only has one member.

```
axiom unit_unique (x y: unit): x = y.
```

Axioms work the same as lemmas except they do not require proofs. Using axioms can simplify some constructions, but it comes at the cost of not being able to use structural induction without adding it explicitly as an axiom, and the risk of introducing inconsistencies. One pitfall is that all EasyCrypt types must be inhabited[5]. Every type `t` has a member `witness<:t>`. This is in stark contrast to proof assistants based on dependent types, where statements are types and the false statements are those that are uninhabited.

2.1.4 Clones

EasyCrypt development are organized into 'theories', which are analogous to modules in other programming languages. They are importable and exportable namespaces dealing with some related concepts. All the basic results about lists are grouped together in the `List` theory.

Many mathematical results are generic over a set of structures. Many results in group theory apply to all groups, or all groups with some additional properties. During early development EasyCrypt dealt with this by having a `Group` theory with a single generic group type that was defined axiomatically using the group axioms. This does not let you use more unique properties of the group in question, however. Different proofs involving different groups might be interested in different additional properties that are not consistent with each other.

A simple way of solving this would be to manually add in all the axioms and lemmas about generic groups into theories dealing with specific groups like the `Int` theory. The problem with this approach is that the generic group theory might get out of sync with the parts that were copied into `Int`. To avoid this EasyCrypt has a built-in concept of cloning a theory, which duplicates the theory when EasyCrypt is ran without modifying the source.

```
theory Foo.
op bar: nat -> nat.
end Foo.
```

[5]In type theory the term 'inhabited' is preferred over 'non-empty' because they are not equivalent in constructive mathematics, with the first being the stronger notion.

```
clone Foo as Baz.
```

In the above snippet we write a theory, `Foo` with a single operator, `bar`, and use cloning to generate a duplicate of it with the name `Baz`. Just duplicating the theory is not very useful, but it is possible to change things in the clone as follows:

```
theory Defaulting.
type t.
op default: t.

op unwrap (x: t option): t =
   with x = None => default
   with x = Some x' => x'.
lemma unwrapK x: unwrap (Some (unwrap x)) = unwrap x by done.
end Defaulting.

clone Defaulting as NatDefault with
 type t <- nat,
  op default <- 0.
```

The `option` type constructor is `type 'a option = [None | Some of 'a].`. It is sometimes used for computations that can fail on sensible inputs. Decryption returns an option since sometimes not all valid ciphertexts can be decrypted. In the code snippet we replace the type, `t`, with the natural numbers and the `default` operator with zero. Then we get the `unwrap` operator and `unwrapK` lemma for free and if the `Defaulting` theory has additional lemmas or operators we will get those as well.

The `NatDefault` theory looks as follows:

```
theory NatDefault.
  op unwrap (x : nat option) : nat =
    with x = None => 0
    with x = Some x' => x'.

  lemma unwrapK:
    forall (x : nat option),
      (unwrap (Some ((unwrap x))) =
      (unwrap x) by done.
end NatDefault.
```

Of note is that we are missing the `t` type and `default` operator. This can be changed by using `<=` or `=` instead of <–, which redefine the instantiated object instead of erasing it.

Usually we wish to impose some restrictions on the operators and types. We do this by stating axioms about them in the base theory. These axioms can then be proven when we do the clone.

```
abstract theory SemiGroup.
type t.
op ( * ): t -> t -> t.
axiom assoc x y z: (x * y) * z = x * (y * z).
end SemiGroup.

clone SemiGroup as BoolSG with
type t <- bool,
op ( * ) <- (&&)
proof *.

realize trans by smt().
```

The theory `SemiGroup` has a type, an operator `( * )` and an axiom stating that the operator is associative. The theory is `abstract`, which prevents access to the internal objects from the outside through any means other than cloning and is used for theories that only exist to be instantiated with more concrete types or operators. The boolean ‘and’ operator has a semigroup structure, so we can clone the theory with booleans and the `&&` operator. To avoid introducing any inconsistencies we use `proof *`, which forces us to prove all the axioms in the instantiated theory. It is also possible to only prove some of the axioms, which is useful when some of the internals of the theory are defined axiomatically. The proof is done on the final line using an smt solver, a powerful proof automation tool.

Although clones are very powerful they are also hard to work with. Although `proof *` can be used to guarantee that a theory is axiom-free and thus that the instantiation of the clone is sound this does not work for theories containing axiomatic definitions, where some axioms are intended to be left unproven. Although many axiomatic constructions can be made axiom-free this is not always possible in EasyCrypt. In Section 2.1.5 we shall see an example of an axiomatic definition that might not be possible to be rewritten without axioms.

Other difficulties arise from the way clones are instantiated. Some theories need to reason about multiple different types of objects, each with their own theory that has to be cloned. This can lead to very long clones, especially when each of the theories have multiple axioms and need multiple operators to be instantiated. The following is an example from our work:

```
clone include MLWE with
  type ZR.t     <- polyXnD1,
   op  ZR.zeror <- zeroXnD1,
   op  ZR.oner  <- oneXnD1,
   op  ZR.(+)   <- PolyReduceZp.(+),
   op  ZR.( * ) <- PolyReduceZp.( * ),
   op  ZR.([-]) <- PolyReduceZp.([-]),
   op  ZR.invr  <- invr,
  pred ZR.unit  <- unit,
   op  duni     <- duni,
   op  dnoise   <- dnoise
  rename [type] "vector" as "zpvector"
  rename [type] "matrix" as "zpmatrix"
remove abbrev ZR.(-)
proof ZR.addrA by exact/ComRing.addrA,
      ZR.addrC by exact/ComRing.addrC,
      ZR.add0r by exact/ComRing.add0r,
      ZR.addNr by exact/ComRing.addNr,
      ZR.oner_neq0 by exact/ComRing.oner_neq0,
      ZR.mulrA by exact/ComRing.mulrA,
      ZR.mulrC by exact/ComRing.mulrC,
      ZR.mul1r by exact/ComRing.mul1r,
      ZR.mulrDl by exact/ComRing.mulrDl,
      ZR.mulVr by exact/ComRing.mulVr,
      ZR.unitP by exact/ComRing.unitP,
      ZR.unitout by exact/ComRing.unitout,
      duni_ll by exact/duni_ll',
      duni_uni by exact/duni_uni',
      duni_funi by exact/duni_funi',
      duni_fu by exact/duni_fu',
      dnoise_ll by exact/dnoise_ll'.
```

Here we instantiate the inner uninstantiated ring theory `ZR` and some distributions.

To simplify this process EasyCrypt allows cloning entire theories instead of just single operators and types. The way this works is that every operator in the inner theory is instantiated with ones with the same name in the theory we instantiate with.

```
theory Bar.
   theory Foo.
      op bar: bool.
      axiom bart: bar.
   end Foo.
   op foo = Foo.bar.
   lemma foot: foo by exact Foo.bart.
end Bar.

theory Baz.
   op bar = true.
   lemma bart: bar by done.
end Baz.

clone Bar as B with
 theory Foo <- Baz.
```

In the above listing we instantiate the `Foo` subtheory of `Bar` with the concrete `Baz` theory. It is not always possible to instantiate with a theory when there is a name mismatch however, and this is exacerbated when there are other operators or types using the names we would like to use. This is the case in the `clone include MLWE` example.

To fix this problem EasyCrypt plans to adopt typeclasses [23], a different method for generalization over theories that is used in some other proof assistants like Coq and Agda that have a longer history, more development and more users. This is relevant to our work because the matrix library should be rewritten once type classes land and might thus look somewhat different in the future.

2.1.5 Subtypes and Quotients

Many mathematical constructions rely on subsets or quotients. The Cauchy reals are a quotient over a subset of the set of rational-valued sequences and the Dedekind reals are a subset of the sets of rationals. It is unclear in general how to replicate these constructions in type theory with function types and inductive definitions alone[6], which motivates subtypes and quotient types.

[6]For the reals this is possible using continued fractions, with rationals being tuples of integers with a 1-terminating list of positive integers and irrationals being tuples of integers with an infinite sequence of positive integers. A caveat of this construction is that some basic operations become uncomputable since we can use the representation to decide whether a

In EasyCrypt both subtypes and quotient types are defined using abstract theories.

```
abstract theory Subtype.
type T, sT.
pred P : T.

op insub : T  -> sT option.
op val   : sT -> T.
op wsT : T.

axiom insubN (x : T): !P x => insub x = None.
axiom insubT (x : T):  P x => omap val (insub x) = Some x.
axiom valP (x : sT): P (val x).
axiom valK: pcancel val insub.
axiom insubW: insub wsT = Some witness<:sT>.
end Subtype.
```

The type `T` is instantiated with the type we wish to subtype and the predicate `P` is instantiated with a predicate determining whether an element of the supertype should be included in the subtype. We get a partial projection function `insub` with an inverse `val`, which can be used to define operators involving the subtype. For soundness we need the subtype to be non-empty. At the time of writing the user is responsible for this, though using the subtype theory adds some axioms which can be validated by a third party.

The `Quotient` theory is defined in terms of the `Subtype` theory, but is instantiated using an equivalence relation on the type we want to take the quotient of and we also do not need to take extra care to ensure that the quotient is non-empty. From an interface standpoint it is very similar to `Subtype`, except that instead of a partial injective projection we get a total surjective projection.

2.2 Hoare Logic

The rules for Hoare Logic and its extensions can get fairly involved, and a comprehensive understanding is not necessary to understand the proofs. We will not present all the rules here. We refer the reader to the EasyCrypt reference manual [1] for the rules implemented in EasyCrypt and to the in-progress book by Gilles Barthe [7], for more details on the underlying theory.

number is equal to 0 or not and computable subtraction could be used to decide equality, which is not decidable for sequences of integers. There is reason to believe that a construction with computable operations is impossible [5]

Cryptography deals intimately with computation, so to work with cryptography we need to model potentially stateful computation. EasyCrypt uses typed Hoare logic to model compuation. Hoare logic reasons about $\{P\}C\{Q\}$-triples where the predicate P is a precondition, the predicate Q a postcondition and C is a program. The predicates can use program variables. A hoare triple is true if assigning values to the variables involved in a way such that P is satisfied and running C results in non-termination or Q being satisfied.

In EasyCrypt programs, also called procedures, are always part of a module, a stateful object consisting of variables and procedures modifying them. The following code defines a module with a single global program variable, `x` of integer type, and a single procedure, `c` that does nothing:

```
module M = {
  var x: int
  proc c() = {
  }
}.
```

The hoare triples in EasyCrypt can only refer to global program variables or the inputs and outputs of the procedures. The following hoare triple says that if the global variable `x` is 0 at the start it will also be 0 if `M.c` terminates; `hoare[M.c: M.x = 0 ==> M.x = 0]`.

We will deal with probabilistic programs later in Section 2.3. The statements in Hoare logic are defined inductively as follows.

$$\begin{array}{rcl} \langle\text{program}\rangle & \models & \texttt{skip} \\ & & \mid\ \langle\text{var}\rangle \leftarrow \langle\text{expression}\rangle \\ & & \mid\ \texttt{if}\ \langle\text{predicate}\rangle\ \texttt{then}\ \langle\text{program}\rangle\ \texttt{else}\ \langle\text{program}\rangle \\ & & \mid\ \texttt{while}\ \langle\text{predicate}\rangle\ \texttt{do}\ \langle\text{program}\rangle \\ & & \mid\ \langle\text{program}\rangle\,\texttt{;}\,\langle\text{program}\rangle \end{array}$$

`skip` is the program that does nothing. It comes with a rule

$$\frac{P \Rightarrow Q}{\{P\}\texttt{skip}\{Q\}}$$

In EasyCrypt any program without contents has the semantics of `skip`(). It can be used as follows:

```
lemma test_skip (P: int -> bool): hoare[M.c: P M.x ==> P M.x].
proof.
```

```
proc.
skip.
trivial.
qed.
```

Here we prove that any predicate on `M.x` that holds before also holds after the computation. The `proc` tactic inlines the procedure, and the `skip` tactic uses the rule for **skip**() to reduce the goals, the set of statements we need to prove to finish the proof, to just `P M.x => P M.x`, which is trivially true.

Program composition, $C_1; C_2$, executes C_1 followed by C_2. We use an intermediate predicate, Q, in the rule as follows:

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

The composition rule is one of the trickier rules to use in practice because it can be hard to automatically infer what the intermediate predicate should be.

We also have an assignment statement, $\langle\text{var}\rangle \leftarrow \langle\text{expr}\rangle$. It can be used to do computations by expressing the computation on the right hand side. $x \leftarrow x+1$ increments 1, for instance. The rule for assigment is

$$\frac{}{\{P[e/x]\}x \leftarrow e\{P\}}$$

where $P[e/x]$ is the predicate with all occurences of x replaced with e. Intuitively, if something is true for e before assignment it is true for x after we assign the expression e to it. This rule is not used directly in EasyCrypt, but is used together with composition in a tactic, `wp` that can simplify hoare statements involving programs with assigments at the end. We demonstrate with a program that negates a boolean program value;

```
module M1 = {
  var x: bool
  proc c() = {
    x <- !x;
  }
}.

lemma test_assign: hoare[M1.c: M1.x ==> ! M1.x].
proof.
proc.
wp.
```

```
skip.
trivial.
qed.
```

Here `wp` produces the goal `hoare[skip : M1.x ==> ! ! M1.x]` (not valid EasyCrypt), which can be solved with `skip`. The `wp` tactic works by using the composition rule to split off the assignment, selecting $P[e/x]$ as the intermediate value, and solving the hoare triple with the assignment using the assignment rule.

The if statement does as one would expect. It uses the logical statement to select between the first and the second computation. Its rule is a bit more complex than the previous:

$$\frac{\{P \wedge B\}C_1\{Q\} \quad \{P \wedge \neg B\}C_2\{Q\}}{\{P\}\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\{Q\}}$$

The rule works by case analysis. To prove a hoare triple involving a conditional we prove both hoare triples corresponding to the two branches, where we get an extra precondition from the conditional and its truth value when the branch is taken.

We demonstrate with a program that checks a program variable to decide whether it should set another one to `true` or `false`.

```
module M2 = {
  var x: bool
  var y: bool
  proc c() = {
    if (x) {
      y <- true;
    } else {
      y <- false;
    }
  }
}.

lemma test_if: hoare[M2.c: true ==> M2.x = M2.y].
proof.
proc.
if.
- auto. smt().
- auto. smt().
qed.
```

The `if` tactic generates two goals, just as in the rule. We solve both goals with the `auto` tactic, which tries to simplify goals involving programs by reducing them to ones with shorter programs. In this case it is able to get rid of the programs entirely, yielding purely logic goals that can be solved using smt solvers.

The while loop executes the inner program repeatedly while the statement is true. Its rule is

$$\frac{\{P \land B\}C\{P\}}{\{P\}\texttt{while}\ B\ \texttt{do}\ C\{P \land \neg B\}}$$

If the inner program preserves P when B holds, then if P holds to begin with then after executing the while loop P will still hold, and B will now not hold. We do not use while loops at all in our main work.

In addition to the rules for the specific program constructs we also have a rule for weakening hoare triples:

$$\frac{P \Rightarrow R \quad \{R\}C\{S\} \quad S \Rightarrow Q}{\{P\}C\{Q\}}$$

This rule is sometimes useful to get the predicates into a certain form.

It is possible to give this language an operational semantics giving an abstract definition of machine state and how programs change it, define hoare triples in terms of this semantics and prove that the rules hold.

Typed Hoare logic comes with the additional restriction that every expression, including the variables, should have a type and assignment is only valid if the expression has the same type as the variable.

2.3 Probabilistic Hoare Logic (pHL)

Basic Hoare Logic only deals with deterministic algorithms. There are extensions to Hoare Logic that allow you to also reason about non-deterministic algorithms, where the program state is a set of possible states, but for cryptography we need to reason about probabilistic algorithms instead since cryptographic theorems often bound the probability of some adversary being able to perform some task or reason that the distribution of some computation is close to uniform.

The semantics of probabilistic programs in the context of cryptography models state as discrete subdistributions. A subdistribution is function to the non-negative reals where the infinite sum is less than 1. Non-deterministic semantics is easier to work with, and is more suitable in situations where events might not have a fixed distribution such as with operations that only occur on an abstract machine or operations that have different behavior on different architectures or when executed in a different context. Modeling this is not useful for cryptographic applications however. In cryptography we at best have an unknown distribution. EasyCrypt models probabilistic computation using pHL.

Probabilistic Hoare Logic extends Hoare Logic by adding pHL statements;

$$\{P\}C\{Q\}\Diamond p,$$

where $\Diamond$ is one of $=$, $\leq$, $\geq$, $<$ or $>$ and p is a real number. A pHL statement is similar to a Hoare statement, except now we do not have certainty of a postcondition holding and instead provide a bounded or exact probability. Note that we cannot just treat $\{P\}C\{Q\}$ as a probability since uninitialized variables are not assumed to follow any specific distibution. While EasyCrypt does tell you if you try to use uninitialized local program variables it does not if you use uninitialized global program variables, and the first check has false positives and only gives a warning.

Instead EasyCrypt also gives you probability statements which evaluate to real numbers but also require an additional 'memory' argument that encodes the initial state the notation for these is as follows; `Pr[M.c() @ &mem : P]` where `mem` is the program memory and `P` is a predicate possibly involving the result of the computation or global variables. The Predicate is a postcondition but it is also possible to refer to the values of variables before the computation by using the memory to refer to the values of the variables in that memory such as in `M.x{mem}`

We also need a new operation to actually generate random output, so we add a program statement which can sample from a subdistribution D and assign the result to a variable x, $x \xleftarrow{\$} D$.

2.4 Probabilistic Relational Hoare Logic (pRHL)

Game based proofs often argue that the behavior of two programs is equivalent. While such statements are possible to formulate in terms of pHL alone they can be made easier to work with by implementing a new type of statements and new rules for them.

These pRHL statements have the syntax `equiv[M.f ~ N.g: P ==> Q]` where `P` and `Q` are formulas involving the program variables. We will only use this to show that two procedures produce equal distributions.

3 Matrix Library

To work with lattice based cryptography we need rectangular matrices and matrix concatenation. The original matrix library in EasyCrypt could only handle square matrices of a fixed size, so this was not supported directly. Some earlier work embedded their matrices into the square matrices [13] but we wanted a more principled approach and thus ended up rewriting the EasyCrypt matrix library to support rectangular matrices of dynamic size. We will detail the design and implementation of our rewrite. The resulting theory can now be found as `DynMatrix.eca` in the EasyCrypt standard library.

3.1 Mathematical foundations

In mathematics the set of n by m matrices over R are defined as $R^{m \times n}$. Formally this means that they are the set of functions from $m \times n$ to R. This uses a formal trick where each natural number n can be defined as a specific set with n elements. This definition is useless on its own, however. We need to be able to define and reason about functions related to matrices.

We take a look at how to formally define matrix multiplication as a function and see how this motivates our definition for matrices in EasyCrypt. We will initially avoid the details of the function and instead focus on its type. In set theory matrix multiplication could be written as a function $\cdot : R^{m \times k} \times R^{k \times n} \to R^{m \times n}$, but this is in reality a family of functions $\cdot_{R,m,n,k}$ where R is a ring and m, n and k are natural numbers. EasyCrypt does not let you reason about families of functions indexed by values from the outside, which presents an issue when we want to multiply and prove things about multiple matrices with different shapes. The way they get around this in HOL [17], which is similar to EasyCrypt is to index with types instead of natural numbers. This matches nicely with the set theoretical definition of matrices since the integers are used as sets there. The type looks something like the following;

$$\cdot_R : (\alpha, \beta, \gamma : \tau) \to (\alpha \times \gamma \to R) \times (\gamma \times \beta \to R) \to (\alpha \times \beta \to R).$$

The matrices are thus defined as $(\alpha, \beta : \tau) \to \alpha \times \beta \to R$. The types, α, β and γ can be inferred from the value arguments and do not have to be provided. This is an elegant definition, but it does not work in EasyCrypt because it does not support concatenation very well and we need concatenation for our work.

The issues with concatenation are easier to illustrate with vectors, R^n than with matrices, though the problems with vectors are analogous to problems with matrices. A sensible definition is $\|_R : R^m \times R^n \to R^{m+n}$, but this does not work for types since addition is not defined for arbitrary types and we need to add n and m in the codomain. The next best thing is to use a type constructor $T : \tau \times \tau \to \tau$ to emulate addition. The type constructor needs to output a type which has a number of elements equal to the sum of the number of element in the two arguments. The disjoint union has this property, but we will show that no choice gives us associativity.

Lemma 1. *There is no choice for the binary type constructor $T : \tau \times \tau \to \tau$ and function $\|_R : (\alpha, \beta : \tau) \to R^\alpha \times R^\beta \to R^{T(\alpha,\beta)}$ such that for all vectors u, v and w the expressions $\|_R(u, \|_R(v, w))$ and $\|_R(\|_R(u, v), w)$ have the same type and the number of elements in $T(\alpha, \beta)$ is the sum of the number of elements in α and β.*

Proof. Let u have type R^α, v have type R^β and w have type R^γ, where α, β and γ are types with one element. Then the left side has type $R^{T(\alpha,T(\beta,\gamma))}$ and the right side has type $R^{T(T(\alpha,\beta),\gamma)}$. Type constructors are injective, so if these types are equal then $T(\alpha, \beta)$ and α are equal, but for the lemma to be true the left side should have 2 elements and the right side 1, so the types cannot be equal. □

As a consequence the statement of associativity of concatenation would not even be valid with the HOL definition. Equality is only defined for equal types. For our work this is not an immediate problem since we do not need associativity of concatenation, but there are others who do. Specifically this was requested by the authors of the paper on the formal verification of Dilithium [6], who made use of our matrix library.

Since the HOL approach of using types instead of natural numbers does not work we need to try a different approach. One idea is to lift the natural number parameters of the family so we get

$$\cdot_R : \mathbb{N}^3 \to \bigcup_{m,n,k \in \mathbb{N}} (R^{m\times k} \times R^{k\times n} \to R^{m\times n}).$$

We cannot get rid of the R parameter, even in set theory, since there is no set of rings. Likewise there is no complete type of rings. Still, this is not much of an issue since matrices with different elements have limited and less interesting interactions.

In some proof assistants, like Coq, you could translate this definition into type theory and implement it. This does not work in EasyCrypt because there is nothing corresponding to the infinite union[7].

A natural substitute for the union is $\bigoplus_{\mathbb{N}\times\mathbb{N}} R \times \bigoplus_{\mathbb{N}\times\mathbb{N}} R \to \bigoplus_{\mathbb{N}\times\mathbb{N}} R$, where $\bigoplus_{\mathbb{N}\times\mathbb{N}} R \subset R^{\mathbb{N}\times\mathbb{N}}$ is the set of functions that are eventually zero, or more formally where the preimage of non-zero elements is finite. There is a natural injection from every component in the union into this set. With this substitute we get some artifacts since multiplication of matrices that are 'too large' are now defined, but this commonplace in proof assistants. Coq defines division such that $\frac{x}{0} = 0$ and natural number subtraction such that $m - n = 0$ when $n > m$. The Coq definitions are still consistent with the normal definitions where the latter are defined. This is formally how it works in set theory as well, though it is usually swept under the rug in practice.

Substituting the union we get

$$\cdot_R : \mathbb{N}^3 \to \bigoplus_{\mathbb{N}\times\mathbb{N}} R \times \bigoplus_{\mathbb{N}\times\mathbb{N}} R \to \bigoplus_{\mathbb{N}\times\mathbb{N}} R.$$

It turns out, however, that this definition is inconvenient to work with in a proof assistant or programming language. The issue is that we need to input the dimensions of the matrix when we do certain operations. For matrix multiplication the dimensions are actually not needed. There is a single function in $\bigoplus_{\mathbb{N}\times\mathbb{N}} R \times \bigoplus_{\mathbb{N}\times\mathbb{N}} R \to \bigoplus_{\mathbb{N}\times\mathbb{N}} R$ that is consistent with all dimension choices. The same is true for most other operators. We do not usually need to know the size of the matrices to work with them. Once again concatenation presents a problem. With concatenation we need to know the 'starting point' of the concatenation, which requires us to know the dimension of the matrix we are working with. Another objection is that without dimension the objects are no longer analogous to matrices. The shape of a matrix is an intrinsic property that should be part of the definition somehow. The easiest way to fix this is to just add the dimension information back to the matrices, defining them as $\mathbb{N}^2 \times \bigoplus_{\mathbb{N}\times\mathbb{N}} R$. A similar construction also allows us to deal with the 'matrices that are too large'-problem by taking the subset where the function is zero for values outside the rectangle defined by the dimension; $M = \{(r, c, f) \in R^{\mathbb{N}\times\mathbb{N}} | \forall i, j \in \mathbb{N}, r \leq i \vee c \leq j \Rightarrow f(i, j) = 0_R\}$.

This, with some minor technical modifications[8], is the definition we ended up using, with the type of matrix multiplication being $M \times M \to M$. There are still shortcomings; multiplication does not require the number of columns

[7]Coq can make this work since it has dependent types, which allow you to reason about families of types parameterized by another type from the outside

[8]EasyCrypt does not define the natural numbers and instead uses the integers by convention

of the first matrix to equal the number of rows of the second, but this was considered the lesser evil compared to the other solutions we have discussed. Much of the implementation work was spent trying to minimize the impact of this shortcoming.

3.2 Implementation

The main challenge of the implementation is how to deal with size mismatch. In our design instead of distinguishing between differently sized matrices at the type level we distinguish between them only at the value level. This means that we need to return *something* for the cases where conventional matrix operations are undefined, like with addition of differently sized matrices. Turning partial function into total ones by returning bogus values for the previously undefined cases is common in proof assistants. In Coq subtraction over the naturals is defined to return zero if the subtracted value is greater. We chose the bogus values in a way that reduces the necessary conditions on sizes for the foundational lemmas to hold. The choices we make in the matrix library are usually specific to the operator in question and will be discussed when we introduce the operator. The most relevant choices are to resize either the arguments or the outputs, or to output a universal bogus value such as the $\times 0$ empty matrix.

We construct matrices as a quotient of tuples $(\mathbb{Z} \times \mathbb{Z} \to R, r : \mathbb{Z}, c : \mathbb{Z})$, where R is a commutative ring, r is the number of rows and c is the number of columns. The equivalence relation considers two matrices the same if the functions are equal in the rectangle defined by r and c. Negative row and column sizes are considered to be 0. Similarly we construct vectors as a quotient of tuples $(f : \mathbb{Z} \to R, s : \mathbb{Z})$.

We will provide an overview of the matrix library. Large parts have been left out or reordered for the sake of exposition.

3.2.1 Vectors

```
clone import Ring.ComRing as ZR.
type R = t.

theory Vectors.

type vector.

type prevector = (int -> R) * int.
```

```
op vclamp (pv: prevector): prevector =
  ((fun i => if 0 <= i < pv.`2 then pv.`1 i else zeror), max 0 pv.`2).

op eqv (pv1 pv2: prevector) = vclamp pv1 = vclamp pv2.

clone import Quotient.EquivQuotient as QuotientVec with
  type T <- prevector,
  type qT <- vector,
  op eqv <- eqv
  proof EqvEquiv.* by smt().
```

We start out by cloning the generic commutative ring theory. After this we can use objects and lemmas from a generic commutative ring. We define a type alias `R` for the cloned commutative ring type. We use the same commutative ring in both the vector and matrix theories, but the first is the vector theory.

To define vectors over `R` we start with a generic type `vector` and a concrete type `prevector`, which we will quotient to get our desired vector type. To perform the quotient we define a function `vclamp` that sends prevectors to the canonical representative in the quotient. In particular it maps sizes below 0 to 0 and function values outside the vector to the neutral element in the ring. Next, `vclamp` is used to define the equivalence relation `eqv` such that elements with the same representative are equivalent.

Now that we have an equivalence relation we can take the quotient. In EasyCrypt this is done by cloning the `Quotient.EquivQuotient` theory and instantiating it with the equivalence relation and the type we are quotienting. We also instantiate it with `vector` to change the name of the resulting quotient type and use the smt solver to prove that `eqv` is an equivalence.

```
op tofunv v = vclamp (repr v).

op offunv pv = pi (vclamp pv).

lemma nosmt vclamp_idemp pv: vclamp (vclamp pv) = vclamp pv.
proof. rewrite /vclamp /#. qed.

lemma nosmt eqv_vclamp pv: eqv pv (vclamp pv).
proof. by rewrite /eqv vclamp_idemp. qed.

lemma tofunvK: cancel tofunv offunv.
proof.
```

```
rewrite /tofunv /offunv /cancel => v; rewrite vclamp_idemp.
by rewrite -{2}[v]reprK -eqv_pi /eqv vclamp_idemp.
qed.

lemma offunvK pv: tofunv (offunv pv) = vclamp pv.
proof. by rewrite /tofunv /offunv eqv_repr vclamp_idemp. qed.
```

In the next part we use the lift `repr` and projection `pi` provided by the quotient theory together with `vclamp` to define a more convenient lift and projection. We prove that the projection is an inverse of the lift and that doing a lift followed by a projection sends us to the canonical representative. The lemmas are straightforward consequences of similar lemmas about the original lift and projection.

```
op size (v: vector) = (tofunv v).`2.

lemma nosmt size_ge0 v: 0 <= size v by smt().

lemma nosmt max0size v: max 0 (size v) = size v by smt().

hint simplify max0size.

lemma size_offunv f n: size (offunv (f, n)) = max 0 n.
proof. by rewrite /size offunvK /#. qed.

hint simplify size_offunv.
```

We define a size operator on vectors that just takes the second element in the canonical representative and prove that the size is non-negative. `max0size` is a simplify hint that gets automatically applied whenever we simplify an expression. Targeted usage of simplify hints can help shorted future proofs.

The final lemma `size_offunv` tells us the size of projected prevectors. This will be used throughout the rest of the theory to prove what sizes the outputs of different operators are. This is also turned into a simplify hint to shorten future proofs.

```
op get (v: vector) (i: int) = (tofunv v).`1 i.

abbrev "_.[_]" (v: vector) (i : int) = get v i.

lemma get_offunv f n (i : int) : 0 <= i < n =>
    (offunv (f, n)).[i] = f i.
```

```
proof. by rewrite /get /= offunvK /vclamp /= => ->. qed.

lemma getv0E v i: !(0 <= i < size v) => v.[i] = zeror by smt().
```

We define an operator `get` that accesses the elements in the vector. We also define an abbreviation to make the notation consistent with other EasyCrypt theories and mathematical notation in general. The abbreviation provides a shorthand `v.[i]` for `get v i`. The next two lemmas state what happens when we use the get operator. The first states what happens when we access the vector within the bounds of its size. The second states what happens when we try to access the vector outside the bounds. This is one of the operators that would be partial in regular mathematics since taking elements from outside the matrix is illdefined, but in EasyCrypt we chose to return the neutral element in those cases. We choose the neutral element of the ring because it interacts nicely with the other operators. As an example getting an element distributes over addition with this choice.

```
lemma eq_vectorP (v1 v2 : vector) : (v1 = v2) <=>
    (size v1 = size v2 /\
     forall i, 0 <= i < size v1 => v1.[i] = v2.[i]).
proof.
split => [->//|[eq_size eq_vi]].
have: tofunv v1 = tofunv v2 by rewrite /tofunv /vclamp /#.
by rewrite -{2}[v1]tofunvK -{2}[v2]tofunvK => ->.
qed.
```

We prove that two vectors are equal if and only if their sizes are equal and all the elements with index between zero and the size are equal. This is very useful and will be the main way to prove equality going forward. The implication is trivial. For the converse we only have to notice that vectors with the same size and same elements have the same canonical representative.

```
op vectc n c = offunv ((fun _ => c), n).

lemma size_vectc n c: size (vectc n c) = max 0 n by done.

hint simplify size_vectc.

lemma get_vectc n c i: 0 <= i < n => (vectc n c).[i] = c by smt(get_offunv).
```

With the foundations of the vector type complete we can start defining some vectors of interest. We start out with constant valued vectors. We prove the obvious lemmas about their size and elements. Every operator that produces vectors will have two such lemmas associated with it to enable the usage of `eq_vectorP` to prove equalities.

```
op zerov n = vectc n zeror.

lemma size_zerov n: size (zerov n) = max 0 n.
proof. by rewrite size_offunv. qed.

lemma get_zerov n i: (zerov n).[i] = zeror.
proof.
case (0 <= i < n) => i_bound; first by rewrite get_offunv.
by rewrite getv0E ?size_zerov /#.
qed.

hint simplify size_zerov, get_zerov.

op emptyv = zerov 0.

lemma size_emptyv: size emptyv = 0 by done.

lemma get_emptyv i: emptyv.[i] = zeror by done.

hint simplify size_emptyv, get_emptyv.

lemma emptyv_unique v: size v = 0 => v = emptyv.
proof. move => size_eq. apply eq_vectorP => /#. qed.
```

Using the constant valued vector we can define the zero valued vector and the empty vector, which is the unique 0×0 vector. In an older iteration of the matrix theory we used the empty vector/matrix as an "error" value for many of the partial operations. This meant that most practical uses that had invalid operations would run into problems when trying to access elements. This would catch some mistakes in definitions and procedures but also place unnecessary burden on the user.

```
    op (+) (v1 v2 : vector) = if size v1 = size v2 then
offunv ((fun i => v1.[i] + v2.[i]), max (size v1) (size v2)).

lemma size_addv v1 v2: size (v1 + v2) = max (size v1) (size v2).
proof. rewrite /(+) size_offunv /#. qed.
```

```
lemma get_addv (v1 v2 : vector) i: (v1 + v2).[i] = v1.[i] + v2.[i].
proof. ··· qed.
```

We define the addition operator on vectors. The elements are the sum of the elements in the two vectors we are adding, and the size is the maximum of their sizes. This is equivalent to zero-padding the vectors until they are the same size and then doing the addition. In the case where their sizes are the same the result will also have the same size. We also prove lemmas that tell us what size and elements the vector resulting from addition has. As mentioned previously we used to use the empty vector for the bad input case, which would require equal sizes as a condition on `get_addv` since the result would be the neutral element regardless of the content of the vectors. One of the disadvantages to zero-padding is that when computing sizes the resulting expressions have a lot of nested `max` which can be hard to deal with, but usually the smt solver will handle this nicely. It also makes some size errors harder to catch, since operations on vectors with mismatched sizes are more likely to yield vectors with the correct elements. Zero-padding also has a negative effect on readability compared to returning the empty vector since it is less clear what the conditions for the 'good path' are, and the logic for the bad path bleeds into the good path logic. Despite these disadvantages, avoiding side conditions was deemed important enough to use zero-padding.

```
op [-] (v: vector) = offunv ((fun i => -v.[i]), size v).

lemma nosmt size_oppv v: size (-v) = size v by done.

lemma getvN (v : vector) i : (-v).[i] = - v.[i].
proof.
case: (0 <= i < size v) => bound; 1: by rewrite /([-]) get_offunv.
by rewrite !getv0E /([-]) //= oppr0.
qed.

hint simplify getvN, size_oppv.
```

We add negation and as usual prove `size_` and `get_` lemmas. In this case the right side of both is clearly simpler than the left side of the equalities and there are no side conditions, so it makes sense to add both lemmas as simplify hints.

```
lemma oppv_zerov n: -(zerov n) = zerov n.
proof. apply eq_vectorP => /= i bound. by rewrite oppr0. qed.
```

```
lemma oppv_emptyv: -emptyv = emptyv by apply oppv_zerov.

hint simplify oppv_zerov, oppv_emptyv.

(* Module-like properties of vectors *)
lemma addvA: associative Vectors.(+).
proof.
move => v1 v2 v3; apply eq_vectorP.
rewrite !size_addv.
split => [/# | i _].
by rewrite !get_addv addrA.
qed.

lemma addvC: commutative Vectors.(+).
proof. ··· qed.

lemma add0v v: zerov (size v) + v = v.
proof.
rewrite eq_vectorP size_addv /= => i bound.
by rewrite get_addv //= add0r.
qed.

hint simplify add0v.

lemma lin_add0v v n: n = size v => zerov n + v = v by subst.
```

After introducing addition and negation we can start doing some arithmetic. The set of vectors of a fixed size form a $\mathbb{Z}$-module. The set of all vectors does not because the additive identity would be the empty vector but there is no way to get the empty vector from an addition involving longer vectors, meaning that there are no additive inverses. This means that we cannot use the vector theory to instantiate the module theory, and instead have to provide equivalents of all the lemmas from that theory ourselves. One way to get a module structure from the set of vectors would be to use the infinite direct sum of the ring. As mentioned this means that there would no longer be a concept of size and we would run into issues with concatenation.

`addvA`, which states that addition of vectors is associative, exemplifies how zero-padding avoids side conditions. This lemma would be true even if we used the empty vector on size mismatch since the empty vector would be absorbing, but the proof would be longer. We would have to deal with every case for how we could get a size mismatch. Commutativity of addition, `addvC`, has a similar story.

The `add0v` lemma exemplifies why we need conditions on some lemmas. It is impossible for all the zero vectors to be a neutral element simultaneously, so we need a condition on the size of the vectors for this. This lemma also showcases a general issue with some proof assistants. For the lemma to be easily readable we write it as `zerov (size v) + v = v`, but this means that the size argument of the zero vector has to be in the correct form, `size v`. Getting the size into the correct form can be inconvenient if it is a long expression since we would have to write out the whole equality that we are trying to rewrite with. To simplify this process we provide a version of the lemma, `lin_add0v` where convertibility into the correct form is a condition instead.

```
(* Inner product *)
op dotp (v1 v2 : vector) =
  bigi predT (fun i => v1.[i] * v2.[i]) 0 (max (size v1) (size v2)).

lemma dotpE v1 v2: dotp v1 v2 =
  bigi predT (fun i => v1.[i] * v2.[i]) 0 (max (size v1) (size v2)) by done.

lemma dotpC : commutative dotp.
proof. ··· qed.
```

We define the dot product using `bigi`, the summation operator from the standard library. As usual we handle size mismatches by zero-padding. It might seem like the `dotpE` lemma is redundant, since you can get the same result by destructuring the defitition. The reason for this redundancy is that there are many operators where we do not want the user to work directly with the definition, so the library tries to provide all the necessary interface through lemmas to make destructuring unnecessary and unidiomatic. As with addition, we can prove unconditional commutativity.

```
lemma dotpDr v1 v2 v3 :
  dotp v1 (v2 + v3) = dotp v1 v2 + dotp v1 v3.
proof. ··· qed.
```

`dotpDr` is an example of a dot product lemma that would need side conditions if we did not do zero-padding. The most natural alternative to zero-padding for dot products is to return the neutral element on a size mismatch. If we had zero-padding neither on the dot product or addition we could have a situation where the first two dot products would yield the neutral element because of a size mismatch, while the third would return something else. This would happen if `v1` and `v3` had the same size different from the size of `v2`. If only the dot product did zero-padding we could get the neutral element from the first dot product and something else from the two on the right side of the equation. This would happen if the size of `v1` was different from the greater of `v2` and `v3`. If only addition did zero-padding the first two dot products could be non-neutral and different, while the third could be neutral because of a size mismatch. This would happen if `v1` and `v2` had the same size, while `v3` was smaller. Interestingly, with no zero-padding we would still not need equality between all the sizes. It would be enough for `v2` and `v3` to have the same size. This would guarantee that either all the dot products are zero because of a size mismatch or dot product with the empty vector or none of them are.

```
(* Vector concatenation *)
op catv (v1 v2: vector) =
  offunv ((fun i => v1.[i] + v2.[i-size v1]), size v1 + size v2).

abbrev ( || ) v1 v2 = catv v1 v2.

lemma size_catv (v1 v2: vector): size (v1 || v2) = (size v1 + size v2).
proof. rewrite /catv /= /#. qed.

lemma get_catv (v1 v2: vector) i :
  (v1 || v2).[i] = if i < size v1 then v1.[i] else v2.[i - size v1].
proof. ··· qed.

lemma get_catv' (v1 v2: vector) i: (v1 || v2).[i] = v1.[i] + v2.[i-size v1].
proof. ··· qed.
```

Concatenation is the reason we cannot use the HOL design for our theory, and part of the reason we do not use the infinite direct sum of the ring. If we used the infinite direct sum we would need to provide the concatenation with an explicit length.

Reading the `get_catv` and `size_catv` lemmas is the easiest way to understand the operator, but this operator is special in that we provide two versions of the `get_catv` lemma. The first makes it easier to understand what the operator does and allows us to use cases to shorten the expression. The second turns directly into arithmetic with no case handling necessary, which simplifies proving some arithmetic lemmas, but can lead to longer and less readable expressions. The second lemma holds because trying to access a value outside the vector yields the neutral element. We also define a subvector operator to reverse concatenation.

```
(* List-vector isomorphism *)
op oflist (s : R list): vector = offunv (nth witness s, size s).

lemma size_oflist l: size (oflist l) = size l.
proof. rewrite size_offunv; 1: smt(List.size_ge0). qed.

lemma get_oflist w (l: R list) i: 0 <= i < size l => (oflist l).[i] = nth w l i.
proof. by move => bound; rewrite get_offunv // (nth_change_dfl w). qed.

op tolist (v : vector): R list = map (fun i => v.[i]) (range 0 (size v)).

lemma size_tolist v: size (tolist v) = size v.
proof. rewrite size_map size_range /#. qed.

lemma nth_tolist w v i: 0 <= i < size v => nth w (tolist v) i = v.[i].
proof. ··· qed.

lemma mem_tolist x v : x \in tolist v <=> exists i, 0 <= i < size v /\ x = v.[i].
proof. by rewrite mapP; smt(mem_range). qed.

lemma oflistK: cancel oflist tolist.
proof. ··· qed.

lemma tolistK: cancel tolist oflist.
proof. ··· qed.
```

Our vectors are isomorphic to the lists in the standard library in a natural way. Both are uniquely defined by an ordered and finite number of elements. This suggests that we could have used lists to define vectors. While this is true it would introduce a lot of noise when doing searches since the list operators are everywhere in the standard library and not all results involving lists are relevant for vectors. It also would not extend very well to matrices as we shall see. Keeping the vector and matrix constructions and lemmas as similar as possible makes the library easier to use.

```
op dvector (d: R distr) (n: int) = dmap (dlist d (max 0 n)) oflist.
```

The final operator in the vector theory is the definition of a distribution over the vectors which samples a size `n` vector elementwise using a given distribution `R` over the base ring. We use the list isomorphism in the definition because lists already have an operator, `dlist`, which does essentially the same thing. Despite this, the lemmas about distributions are some of the longest in the library, especially once we get to matrices.

3.2.2 Matrices

The construction and additive arithmetic of matrices is very similar to vectors. One difference is the fact that, unlike vectors where we could have used lists instead, we are forced to use quotients or subtypes here. This is because lists of lists only have a global length in one direction, the length of the outer list. The lengths of the inner lists might be different, and in the case where the outer list is empty there will be no list to take the length of at all. We could have solved the problem of inner lists with different lengths by taking a quotient or subtype of the list of lists, but we would still run into problems with an empty outer list and as we will see there are other reasons for not making this choice.

```
lemma size_offunm f r c: size (offunm (f, r, c)) = (max 0 r, max 0 c) by done.

abbrev mrange m (i j : int) = 0 <= i < rows m /\ 0 <= j < cols m.

lemma get_offunm f r c (i j : int) : mrange (offunm (f, r, c)) i j =>
  (offunm (f, r, c)).[i, j] = f i j.
proof. rewrite /get /= /mclamp /= /#. qed.
```

We define a `mrange` abbreviation to simplify expressions. It is a predicate that says that the provided indices are inside the bounds of the matrix. Much of the rest of the matrix theory is similar to the vector theory.

```
(* Matrix with the values of v on the diagonal and zeror off the diagonal *)
op diagmx (v : vector) =
  offunm ((fun i j => if i = j then v.[i] else zeror), size v, size v).

lemma get_diagmx v i j: (diagmx v).[i, j] = if i = j then v.[i] else zeror.
proof. ··· qed.

abbrev diagc n (c : R) = diagmx (vectc n c).

abbrev onem n = diagc n oner.
```

The first matrix operator with no vector analogue is the `diagmx` operator, which takes a vector and yields the matrix with diagonal equal to that vector and zeros off the diagonal. We use this to define the identity matrix `onem` as the matrix with ones on the diagonal.

```
(* matrix transposition *)
op trmx (m : matrix) = offunm (fun i j => m.[j, i], cols m, rows m).
```

The transpose is one of the main reasons for defining matrices the way they are defined. A list of lists is not a very symmetric structure. Transposes allow us to exploit the symmetry of matrices to get a lot dual results almost for free. A common pattern for the matrix theory is that we define an operator, prove a bunch of lemmas about it and then define the dual operator, prove that it is dual and then prove the dual lemmas using the duality.

```
(* Gets the n-th row of m as a vector *)
op row m n = offunv (fun i => m.[n, i], cols m).

lemma rowD m1 m2 n: row (m1 + m2) n = row m1 n + row m2 n.
proof. ··· qed.

(* Gets the n-th column of m as a vector *)
op col m n = offunv (fun i => m.[i, n], rows m).

lemma row_trmx m n: row (trmx m) n = col m n by rewrite eq_vectorP.

lemma col_trmx m n: col (trmx m) n = row m n by rewrite eq_vectorP.

hint simplify row_trmx, col_trmx.

lemma colD m1 m2 i: col (m1 + m2) i = col m1 i + col m2 i.
proof. smt(trmxK col_trmx trmxD rowD). qed.
```

As an example of the pattern we define a `row` operator, which yields the n-th row of the matrix as a vector and prove that it distributes over addition. The proof is 4 lines. Then we define the dual `col` operator, which yields the n-th column, prove that taking a row of the transpose is the same as taking a column of the original and vice versa, showing that they are dual. Finally we prove that the column operator distributes over addition by using the fact that transposes are involutions, the fact that the row and column operators are dual under transposition and the distibutivity of the row and transpose operators.

```
(* Matrix multiplication *)
op ( * ) (m1 m2 : matrix) =
  offunm (fun i j => dotp (row m1 i) (col m2 j), rows m1, cols m2).
```

```
lemma rows_mulmx m1 m2: rows (m1 * m2) = rows m1.
proof. move => cols_eq; by rewrite /( * ) cols_eq. qed.

lemma cols_mulmx m1 m2: cols (m1 * m2) = cols m2.
proof. move => cols_eq; by rewrite /( * ) cols_eq. qed.

lemma get_mulmx m1 m2 i j: (m1 * m2).[i,j] = dotp (row m1 i) (col m2 j).
proof. ··· qed.
```

Matrix multiplication is one of the most important parts of the library and is defined using the dot product. The relevant lemmas are unconditional because of the unconditionality of the dot product lemmas.

```
(* Old code not included in the library *)
lemma mulmxDl (m1 m2 m : matrix) :
  size m1 = size m2 => (m1 + m2) * m = (m1 * m) + (m2 * m).
proof. ··· qed.

lemma mulmxA m1 m2 m3: cols m1 = rows m2 => cols m2 = rows m3 =>
  m1 * (m2 * m3) = (m1 * m2) * m3.
proof. ··· qed.
```

As another demonstration of the benefits of zero padding we show older versions of the `mulmxDl` and `mulmxA` lemmas from when matrix multiplication and addition returned the empty matrix on size mismatch. Size mismatch for matrix multiplication is when the first matrix has number of columns unequal to the number of rows of the second matrix.

`mulmxDl` failed to be unconditional because multiplying a 0 by 0 matrix with a 0 by n matrix would yield a 0 by n matrix, allowing us to do arithmetic with the empty matrix and end up with a non-empty matrix at the end. In particular the case where the two matrices we are adding have different sizes failed when the matrix we multiply with has 0 rows. We could still be unconditional on the matrix we were multiplying with, however.

Another lemma with a similar issue was `mulmxA`. If `m1` is n by n, `m2` is m by m and `m3` is 0 by m for $n \neq m$, then multiplying together the first two first yields a 0 by m matrix while multiplying the last two first yields the 0 by 0 matrix. The only size mismatch here is between the first and second matrices, demonstrating that we need a condition relating their sizes. A similar counterexample demonstrates the same for the second and third matrices.

With zero padding matrix multiplication and addition act similarly when there is a size mismatch and when there is not. The difficulties with zero padding occur because of the matrix size, not the contents, but dealing with size is usually easier than arguing about the contents.

```
(* Turns row vector into matrix *)
op rowmx (v: vector) = offunm ((fun _ i => v.[i]), 1, size v).

lemma size_rowmx v: size (rowmx v) = (1, size v) by done.

lemma get_rowmx v i: (rowmx v).[0,i] = v.[i].
proof. ··· qed.

lemma rowK v: row (rowmx v) 0 = v by rewrite eq_vectorP.
```

There is a correspondence between 1 by n matrices and size n vectors. We have already seen one half of this correspondence in the form of `row _ 0`, yielding the first row of a matrix as a vector, and `rowmx` gives the second half, turning a vector into a matrix. We prove that `row _ 0` is a left-inverse of `rowmx`. This correspondence brings up some questions. One might imagine avoiding vectors altogether and just working with matrices, or letting vectors be defined in terms of matrices. We have avoided the first because the distinction can be meaningful, and working with a more general theory can carry unnecessary baggage when it is used. We have avoided the second because of conflicting goals. We want a clean separation between the matrix and vector theory for readability and maintainability and we want to use vectors in the matrix theory in multiple spots, like matrix multiplication and the `row` and `diagmx` operators. Defining vectors in terms of matrices would (in EasyCrypt) necessitate putting the matrix theory first, but this would prevent us from using vectors in the matrix theory.

```
(* Matrix and vector multiplication *)
op mulmxv m v = col (m * colmx v) 0.

abbrev ( *^ ) (m : matrix) (v : vector) : vector = mulmxv m v.

lemma mulmxvE m v: m *^ v = col (m * colmx v) 0 by done.

lemma size_mulmxv m (v: vector): size (m *^ v) = rows m by done.

lemma get_mulmxv m v i: (m *^ v).[i] = dotp (row m i) v.
proof. by rewrite mulmxvE /= get_mulmx. qed.
```

We use the correspondence from the previous paragraph to define matrix vector multiplication by lifting the vector to a matrix and then doing matrix multiplication instead. In regular mathematics this would be convoluted, but in a proof assistant it is desirable to use existing theory whenever possible since the cost of writing entirely new proofs is so high.

```
(* Sideways matrix concatenation - aka row block matrices *)
op catmr (m1 m2: matrix) =
  offunm ((fun i j => m1.[i, j] + m2.[i, j-cols m1]),
          max (rows m1) (rows m2), cols m1 + cols m2).

abbrev ( || ) m1 m2 = catmr m1 m2.

lemma col_catmrL m1 m2 i: rows m1 = rows m2 => i < cols m1 =>
  col (m1 || m2) i = col m1 i.
proof. ··· qed.

(* Downwards matrix concatenation - aka column block matrices *)
op catmc (m1 m2: matrix) =
  offunm ((fun i j => m1.[i, j] + m2.[i-rows m1, j]),
          rows m1 + rows m2, max (cols m1) (cols m2)).

abbrev ( / ) m1 m2 = catmc m1 m2.

op subm (m: matrix) (r1 r2 c1 c2: int) =
  offunm ((fun i j => m.[i+r1,j+c1]), r2-r1, c2-c1).
```

We define matrix concatenation. Unlike vectors there are two dual ways to do the concatenation. As usual transposes simplify the work of proving lemmas about the dual. Unlike concatenation, the partial inverse of taking a sub-matrix does not need two versions. It is convenient to deal with both directions using one operator.

Matrix concatenation also provides an example of a lemma, `col_catmrL`, that has a side condition that we cannot get rid of with zero padding. The condition `rows m1 = rows m2` is necessary because otherwise the result of the concatenation might have more rows than `m1`, leading to a size mismatch. Strictly speaking we only need `rows m2 <= rows m1`, but working with equalities is easier than working with inequalities.

The final part of the matrix theory is, as with vectors, about distributions. These are, by far, the longest and most complicated lemmas in the theory, while simultaneously being entirely trivial to a human. In fact, all the lemmas in the theory are entirely trivial. Still, as we have seen, there are some tricky edge cases, especially if the goal is to provide as many unconditional lemmas as possible.

4 Lattice Based Cryptography

Lattice based cryptograph is the branch of cryptography relying on hard lattice problems such as the Shortest Vector Problem (SVP) and Learning with Errors (LWE) problem. It is attractive because these problems are conjectured to be hard, even for quantum computers, and because lattices turn out to be great building blocks for Homomorphic Encryption, which facilitates computation on encrypted data.

While we will not do Homomorphic Encryption we will define a lattice problem, build a cryptosystem on top of it, and prove it conditionally secure in EasyCrypt.

4.1 MLWE

The Module Learning with Errors (MLWE) problem is a generalization of the LWE problem first introduced by Regev [20] and the Ring Learning with Errors (RLWE) problem. The generalization was first introduced by Brakerski Gentry and Vaikuntanathan [10] as Generalized Learning with Errors (GLWE). It has since then been generalized further, but we will consider a formulation similar to the one in the BGV paper. We will use the decisional variant DMLWE.

Definition 1 (MLWE distribution)**.** *Let ϕ and θ be distributions over a ring R and m a positive integer. The* $\mathrm{MLWE}_{R,m,n,\phi,\theta}$ *distribution is* $(\mathbf{A}, \mathbf{A}\mathbf{s}+\mathbf{e})$ *where* $\mathbf{A} \leftarrow\$ R^{m\times n}$, $\mathbf{s} \leftarrow\$ \theta^n$ *and* $\mathbf{e} \leftarrow\$ \phi^m$.

Definition 2 (DMLWE problem)**.** *The* $\mathrm{DMLWE}_{R,m,n,\phi,\theta}$ *problem is to distinguish between the* $\mathrm{MLWE}_{R,m,\phi,\theta}$ *distribution and a uniformly sampled tuple.*

We model this as a game in EasyCrypt. We uniformly randomly select a bit `b <$ {0,1}` and use this to decide whether we provide the adversary with a uniformly or MLWE distributed tuple. At the end the adversary has to guess how the tuple was sampled.

In practice this game is only used as an assumption and we instead use the relationship between the advantage of an adversary against this problem and the difference in the advantage of an adversary against the two problems where `b` is fixed. The assumption that the DMLWE problem is hard means we can move between these two games while keeping the advantage of the adversary small.

```
module DMLWE'(A:AdvMLWE) = {
  proc main(ds, r, c) : bool = {
    var b, b', mA, s, e, u;
    b <$ {0,1};
    mA <$ dmatrix duni r c;
    if (b) {
      s  <$ ds;
      e  <$ dvector dnoise r;
      u  <-  mA *^ s + e;
    } else {
      u  <$ dvector duni r;
    }
    b' <- A.guess(mA, u);
    return (b = b');
  }
}.
```

Listing 1: EasyCrypt definition of the DMLWE problem

The module is a game parameterized by the noise distribution `dnoise`, the secret distibution `ds` and the matrix shape `r, c`.

4.2 MLWE based Cryptosystem

We use a cryptosystem described by Lyubashevsky [16] since this avoids the use of the leftover hash lemma, which would complicate the proof. Instead we have to use two instances of the DMLWE problem with differently sized vectors and matrices.

Let $\|x\| := \min(x, p - x)$ be the distance from 0 to x in $\mathbb{Z}_p$. We work in the polynomial ring $R = \mathbb{Z}_p / < \mathbf{x}^n - 1 >$. χ is a distribution over this ring such that the coefficients are sampled uniformly from the interval $[-\beta, \beta]$, in EasyCrypt we use the identifier `dnoise` for this.

We use sampling from the empty subdistribution to signal an error in the encryption process because the EasyCrypt interface for encryption is not fallible. Encryption fails if we get the wrong number of bits to encrypt.

`bits_to_poly` turns the bitstring into a polynomial in the same way the summation in KGen does.

The decryption uses the secret to get rid of the inner product and attempts to retrieve the plaintext from the coefficients. We will see why this works in the correctness proof.

```
KGen
    $\mathbf{s}, \mathbf{e}_1 \leftarrow\$ \chi^m$
    $\mathbf{A} \leftarrow\$ R^{m \times m}$
    $\mathsf{pk} \leftarrow (\mathbf{A}, \mathbf{t} := \mathbf{As} + \mathbf{e}_1)$
    $\mathsf{sk} \leftarrow \mathbf{s}$
    return $(\mathsf{pk}, \mathsf{sk})$
```

```
proc kg() = {
    var pk, sk, mA, t, s, e1;
    s  <$ dvector dnoise m;
    e1 <$ dvector dnoise m;
    mA <$ dmatrix duni m m;
    t  <- mA *^ s + e1;
    sk <- s;
    pk <- (mA, t);
    return (pk, sk);
}
```

```
$\mathsf{Enc}_{\mathsf{pk}}(\mu : \{0,1\}^n)$
    $\mathbf{r}, \mathbf{e}_2 \leftarrow\$ \chi^m$
    $e_3 \leftarrow\$ \chi$
    $\mathbf{u} \leftarrow \mathbf{A}^T\mathbf{r} + \mathbf{e}_2$
    $v \leftarrow \langle \mathbf{r}, \mathbf{t} \rangle + e_3 + \left\lfloor \frac{p}{2} \right\rfloor \sum_{i=0}^{n-1} \mu_i \mathbf{x}^i$
    return $(\mathbf{u}, v)$
```

```
proc enc(pk:pkey, pt:ptext) = {
    var r, e2, e3, mA, t,  u, v;
    var x: bool;
    if (size pt <> n) {
            x <$ dnull;
    }
    (mA, t) <- pk;
    r  <$ dvector dnoise m;
    e2 <$ dvector dnoise m;
    e3 <$ dnoise;
    u <- r ^* mA + e2;
    v <- dotp r t + e3 +
      Zp.inzmod (p %/ 2) ** bits_to_poly pt;
    return (u,v);
}
```

```
$\mathsf{Dec}_{\mathsf{sk}}(\mathbf{u}, v)$
    $d \leftarrow v - \langle \mathbf{u}, \mathbf{s} \rangle$
    return $decode(d)$
```

```
proc dec(sk:skey, c:ctext) = {
    var u, v, d;
    (u, v) <- c;
    d <- v - (dotp u sk);
    return Some (decode d);
}
```

```
decode(d)
    return $decodecoeff(d_1), \ldots, decodecoeff(d_n)$
```

```
    op decode d: bool list = map
      (fun i => decode_coeff d.[i]) (range 0 n).
```

```
decodecoeff(d)
// if $d$ is closer to 0 return 0
if $\|d\| < \left\|d - \frac{p}{2}\right\|$
    return 0
// if $d$ is closer to $\frac{p}{2}$ return 1
return 1
```

```
op decode_coeff d = distmodp d
  (Zp.inzmod (p %/ 2)) <= distmodp d Zp.zero.
```

The first noise, e_1, is added to make the public key computationally indistinguishable from uniform, while the two others, $\mathbf{e}_2$ and e_3, hide the ciphertext.

4.2.1 Termination

We prove in EasyCrypt that key generation and decryption terminates and that encryption terminates when the plaintext has the correct length. This is necessary since any postcondition is provable if a procedure cannot terminate.

```
lemma kg_ll: islossless LWEpke.kg.
proc.
auto => />.
rewrite dvector_ll. rewrite dnoise_ll.
rewrite dmatrix_ll // duni_ll.
qed.

lemma enc_valid_ll: phoare[LWEpke.enc: size arg.`2 = n ==> true] = 1%r.
proc.
rcondf 1 => //.
auto => />.
by rewrite dvector_ll dnoise_ll.
qed.

lemma dec_ll: islossless LWEpke.dec by proc; auto.
```

EasyCrypt functions are total and none of the procedures have any loops, so we only need to show that the sampling is lossless, meaning that it is a proper distribution with sum 1. For encryption we also have to argue that we never sample from the empty subdistribution if the input length is correct, which is trivial.

The `dvector_ll` and `dmatrix_ll` lemmas state that sampling element wise is lossless if the sampling we use is lossless. The `dnoise_ll` and `duni_ll` lemmas state that these two distributions are lossless.

4.3 Correctness

For the cryptosystem to be useful we want both security and correctness; that decrypting a valid ciphertext yields the message used to generate it. We will display both a paper proof and its transliteration into EasyCrypt, which will highlight the amount of work that goes into writing these proofs.

4.3.1 Paper Proof

We show that encrypting and decrypting returns the original bit, assuming our parameters are good.

In lattice based cryptography correctness is all about managing the size of the noise. We compute $d = \langle \mathbf{r}, \mathbf{e}_1 \rangle + e_3 + \lfloor \frac{p}{2} \rfloor (\sum_{i=0}^{n-1} \mu_i \mathbf{x}^i) - \langle \mathbf{e}_2, \mathbf{s} \rangle$. Each coefficient is closer to to $\frac{p}{2}\mu_i$ if the total noise, $\|\langle \mathbf{r}, \mathbf{e}_1 \rangle + e_3 - \langle \mathbf{e}_2, \mathbf{s} \rangle\|$, is smaller than $\frac{p}{4}$, which holds when $2mn\beta^2 + \beta < \frac{p}{2}$.

The formal proof follows the same outline, but in more detail, and also needs a few helping lemmas for the distance function.

4.3.2 EasyCrypt proof

We instrument the original system with global variables tracking the total noise.

```
module LWEpke' = {
    var er : polyXnD1
    var r  : zpvector
    var e1 : zpvector
    var e2 : zpvector
    var e3 : polyXnD1
    var s  : zpvector

    proc kg() = {
        var pk, sk, mA, t;
        s  <$ dvector dnoise m;
        e1 <$ dvector dnoise m;
        mA <$ dmatrix duni m m;
        t  <- mA *^ s + e1;
        sk <- s;
        pk <- (mA, t);
        return (pk, sk);
    }

    proc enc(pk:pkey, pt:ptext) = {
        var  mA, t,  u, v;
        var x: bool;
        if (size pt <> n) {
            x <$ dnull;
        }
        (mA, t) <- pk;
        r  <$ dvector dnoise m;
        e2 <$ dvector dnoise m;
        e3 <$ dnoise;
        u <- r ^* mA + e2;
        v <- dotp r t + e3 + Zp.inzmod (p %/ 2) ** bits_to_poly pt;
        return (u,v);
    }

    proc dec(sk:skey, c:ctext) = {
        var u, v, d;
        (u, v) <- c;
        (* Here we compute the total error *)
        er <- dotp r e1 + e3 - (dotp e2 s);

        d <- v - (dotp u sk);
        return Some (decode d);

    }
}.
```

This makes it easier to reason about the noise, since in normal execution the variables that make up the noise are local to their own function call and are lost after the call is complete. It also allows us to state the correctness lemma conditionally on the size of the noise.

By itself this module is questionable at best. Storing global variables means that you can do things like store the plaintext in a global variable and give it back when you "decrypt". For this reason we wish to show correctness for the initial cryptosystem, not the instrumented one. To make this possible we show that the instrumented cryptosystem is correct if and only if the original is.

```
equiv LWEpke_LWEpke'_equiv :
Correctness(LWEpke).main ~ Correctness(LWEpke').main : ={m} ==> ={res} by sim.
```

`Correctness` is a parametric module in the standard library which plays a correctness game. It is provided a message which is encrypted and decrypted, and outputs true iff the decryption equals the original. The equivalence states that both the instrumented and original cryptosystem have the same distribution of results in the correctness game if provided the same message. `sim` is a tactic that uses equality of the variables involved in the two programs. Since the only difference between the two programs is whether the variables are public or not and a computation that is not used for the output (computing `er`), and every variable is set before it is used, meaning the state of uninitialized memory cannot change the behavior, the computations are trivially equal.

```
hoare LWEpke'_small_error_correct : Correctness(LWEpke').main:
    true ==> distpoly LWEpke'.er zeroXnD1 < p %/ 4 => res.
proof.
...
qed.
```

The proof is split into two parts. First we prove that we get the correct value for `d` and that a program which reaches that point got an input of the correct length. The first is a matter of doing simple arithmetic, the second is a matter of noticing that a program where the input has the wrong length almost immediately aborts by sampling the empty subdistribution.

The second half of the proof shows that this value of `d` gives the correct decryption when the error is small. We handle each bit of the message separately using `eq_from_nth`, which shows that two lists are equal if they have the same length and equal elements. We also handle the case where the i-th bit is true and false separately. The proof is also simple arithmetic, but the arithemtic involves the operator `distmodp`, which computes the shortest distance between two elements in $\mathbb{Z}_p$. This metric is not a standard operator in EasyCrypt so we define it and prove some helper lemmas that we use in this proof.

```
op distmodp (a b : Zp) = min (`|asint a - asint b|) (p - `|asint a - asint b|).
```

The definition is fairly natural. `asint` allows us to lift the elements in $\mathbb{Z}_p$ to integers in the interval $[0, p)$, and computing the absolute value of the difference gives us the length of a path between the two values on this interval. In $\mathbb{Z}_p$ this corresponds to the path that does not cross zero. The length of the path crossing zero is `p - `|asint a - asint b|`, and taking the minimum of these gives us the shortest distance between two elements in $\mathbb{Z}_p$. EasyCrypt does not have a theory for metrics, so we have to manually prove the properties we need to use.

```
lemma mod_exists a q: exists m, a %% q = a - m * q by smt(divzE).

lemma abs_asint a: `|asint a| = asint a by smt(rg_asint).

lemma asint_inzmod_half: asint (inzmod (p %/ 2)) = p %/ 2.
proof. rewrite inzmodK. smt(ge2_p). qed.
```

We start by proving some short helper lemmas that are automatic.

```
lemma add_asint a c: exists b, (b = 0 \/ b = 1) /\
  asint a + asint c = asint (a + c) + b * p.
proof.
...
qed.

lemma sub_asint a c: exists b, (b = 0 \/ b = 1) /\
  asint a - asint c = asint (a - c) - b * p.
proof.
...
qed.
```

We prove that adding after lifting instead of lifting after adding can only increase our result by p or have the same result. As a corollary we get a similar result for subtraction.

```
lemma distmodp_zero a b: distmodp a b = distmodp (a - b) zero.
proof.
rewrite /distmodp zeroE /=.
elim (sub_asint a b) => c [[-> | ->] /= H] //=; 1:by rewrite H.
have rel_asint: asint a - asint b < 0.
  smt(rg_asint).
rewrite StdOrder.IntOrder.ltr0_norm // H /= abs_asint /#.
qed.
```

We prove that the distance between two points is the same as the distance from their difference to zero. This is an obvious property we want a metric to have, and it will be useful in future proofs. We use `sub_asint` to reduce the problem to split apart the cases where the subtraction underflows and when it does not. The second case is trivial. The first case requires some more work.

```
lemma dist_asint_low a: distmodp a zero < p %/ 4 =>
        distmodp a zero < distmodp a (inzmod (p %/ 2)).
```

```
proof.
rewrite /distmodp zeroE /= abs_asint {1}/min.
case (asint a < p - asint a) => H1;
rewrite /min H1 /= asint_inzmod_half; smt(rg_asint).
qed.
```

dist_asint_low is a technical lemma required for the correctness proof. It says that elements whose distance to zero can be bounded by $\frac{p}{4}$ are closer to zero than to $\frac{p}{2}$. We need this because whether the an element is closer to zero or to $\frac{p}{2}$ tells us how to decrypt.

```
lemma asint_neg (a: Zp): a <> zero => asint (-a) = p - asint a.
proof. ··· qed.

lemma dist_neg (a b: Zp): distmodp (-a) (-b) = distmodp a b.
proof. ··· qed.
```

The final lemma we need is that the distance between additive inverses is the same as the distance between the originals. To prove this we use the helper lemma `asint_neg`, which tells us what happens to inverses when applying the `asint` operator. We have to handle the zero case separately.

```
lemma LWEpke'_correct_bounded_error &m pt :
    Pr[Correctness(LWEpke').main(pt) @ &m : distpoly LWEpke'.er zeroXnD1 < p %/ 4] <=
    Pr[Correctness(LWEpke').main(pt) @ &m : res].
proof. ··· qed.

lemma LWEpke_correct_bounded_error &m pt :
    Pr[Correctness(LWEpke').main(pt) @ &m : distpoly LWEpke'.er zeroXnD1 < p %/ 4] <=
    Pr[Correctness(LWEpke).main(pt) @ &m : res].
proof.
have ->: Pr[Correctness(LWEpke).main(pt) @ &m : res] =
  Pr[Correctness(LWEpke').main(pt) @ &m : res]
  by byequiv (LWEpke_LWEpke'_equiv).
exact/LWEpke'_correct_bounded_error.
qed.
```

Finally we use the correctness lemma about the instrumented cryptosystem to bound the probability of correct decryption from below and use the equivalence to the real cryptosystem to get a similar bound for that. We could go one step further and get bounds for parameters that guarantee correct decryption, but in lattice cryptosystems it can make sense to allow for a negligible chance of failure to get higher efficiency.

```
IND-CPA
1 :   $\mathbf{s}, \mathbf{e}_1 \leftarrow\$ \chi^m$
2 :   $\mathbf{A} \leftarrow\$ R^{m \times m}$
3 :   $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e}_1$
4 :   $\mathsf{pk} \leftarrow (\mathbf{A}, \mathbf{t})$
5 :   $\mathsf{sk} \leftarrow \mathbf{s}$
6 :   $\mu \leftarrow\$ \{0,1\}^n$
7 :   $\mathbf{r}, \mathbf{e}_2 \leftarrow\$ \chi^m$
8 :   $e_3 \leftarrow\$ \chi$
9 :   $\mathbf{u} \leftarrow \mathbf{A}^T \mathbf{r} + \mathbf{e}_2$
10 :  $v \leftarrow \langle \mathbf{r}, \mathbf{t} \rangle + e_3 + \left\lfloor \frac{p}{2} \right\rfloor \sum_{i=0}^{n-1} \mu_i \mathbf{x}^i$
11 :  $\mu' \leftarrow \mathcal{A}(\mathbf{u}, v)$
12 :  $\mathbf{return}\ \mu = \mu'$
```

```
Game$_1$
1 :   $\mathbf{s}, \mathbf{e}_1 \leftarrow\$ \chi^m$
2 :   $\mathbf{A} \leftarrow\$ R^{m \times m}$
3 :   $\mathbf{t} \leftarrow R^m$
4 :   $\mathsf{pk} \leftarrow (\mathbf{A}, \mathbf{t})$
5 :   $\mathsf{sk} \leftarrow \mathbf{s}$
6 :   $\mu \leftarrow\$ \{0,1\}^n$
7 :   $\mathbf{r}, \mathbf{e}_2 \leftarrow\$ \chi^m$
8 :   $e_3 \leftarrow\$ \chi$
9 :   $\mathbf{u} \leftarrow \mathbf{A}^T \mathbf{r} + \mathbf{e}_2$
10 :  $v \leftarrow \langle \mathbf{r}, \mathbf{t} \rangle + e_3 + \left\lfloor \frac{p}{2} \right\rfloor \sum_{i=0}^{n-1} \mu_i \mathbf{x}^i$
11 :  $\mu' \leftarrow \mathcal{A}(\mathbf{u}, v)$
12 :  $\mathbf{return}\ \mu = \mu'$
```

Figure 6: Using DMLWE to hop to a game where the public key is uniformly random. The advantage of an adversary against the distinguishing game is bounded by the advantage of a related adversary against the DMLWE game. This works because the adversary can use a DMLWE challenge $(\mathbf{t})$ to construct a challenge of the distinguishing game by following the IND-CPA game and Game 1 for everything except the sampling of $(\mathbf{t})$, which is the DMLWE challenge.

4.4 Security

We prove IND-CPA security of the cryptosystem by game hopping, a proof technique that shows indistinguishability of two games by showing indistinguishability of a sequence of games with these two at the ends. In this case we use it to show that the indistinguishability game with a given adversary is indistinguishable from a coin toss. The first hop of the paper proof is shown in Figure 6.

The jump between game 1 and game 1.5 is purely algebraic. In EasyCrypt we handle this jump and the jump from game 1.5 to game 2 at the same time. This is not necessarily the best way of doing things, but simplifying a proof after completion loses some of its value in EasyCrypt since the proofs are unfit for exposition anyways.

```
Game$_1$
1 :  $\mathbf{s}, \mathbf{e}_1 \leftarrow\$ \chi^m$
2 :  $\mathbf{A} \leftarrow\$ R^{m \times m}$
3 :  $\mathbf{t} \leftarrow R^m$
4 :  $\mathsf{pk} \leftarrow (\mathbf{A}, \mathbf{t})$
5 :  $\mathsf{sk} \leftarrow \mathbf{s}$
6 :  $\mu \leftarrow\$ \{0,1\}^n$
7 :  $\mathbf{r}, \mathbf{e}_2 \leftarrow\$ \chi^m$
8 :  $e_3 \leftarrow\$ \chi$
9 :  $\mathbf{u} \leftarrow \mathbf{A}^T \mathbf{r} + \mathbf{e}_2$
10 : $v \leftarrow \langle \mathbf{r}, \mathbf{t} \rangle + e_3 + \lfloor \frac{p}{2} \rfloor \sum_{i=0}^{n-1} \mu_i \mathbf{x}^i$
11 : $\mu' \leftarrow \mathcal{A}(\mathbf{u}, v)$
12 : return $\mu = \mu'$
```

```
Game$_{1.5}$
1 :  $\mathbf{s}, \mathbf{e}_1 \leftarrow\$ \chi^m$
2 :  $\mathbf{A}' \leftarrow\$ R^{m \times (m+1)}$
3 :
4 :  $\mathsf{pk} \leftarrow \mathbf{A}'$
5 :  $\mathsf{sk} \leftarrow \mathbf{s}$
6 :  $\mu \leftarrow\$ \{0,1\}^n$
7 :  $\mathbf{r} \leftarrow\$ \chi^m$
8 :  $\mathbf{e} \leftarrow\$ \chi^{m+1}$
9 :  $\mathbf{u}' \leftarrow \mathbf{A}'^T \mathbf{r} + \mathbf{e} + \mathbf{0}^m || \lfloor \frac{p}{2} \rfloor \sum_{i=0}^{n-1} \mu_i \mathbf{x}^i$
10 :
11 : $\mu' \leftarrow \mathcal{A}(\mathbf{u}')$
12 : return $\mu = \mu'$
```

Figure 7: This step is purely algebraic by $\mathbf{A}' = \mathbf{A}||\mathbf{t}$ and $\mathbf{u}' = \mathbf{u}||v$

```
module RedLWEAdv: AdvMLWE = {
  proc guess(mA, t) : bool = {
    var m0, m1, b, b', r, e2, e3, pt;
    var x: bool;
    (m0, m1) <@ A.choose(mA, t);
       b     <$ {0,1};
       pt    <- if b then m1 else m0;
       if (size pt <> n) {
         x <$ dnull;
       }
       r     <$ dvector dnoise m;
       e2    <$ dvector dnoise m;
       e3    <$ dnoise;
       b'    <@ A.guess(r ^* mA + e2, (dotp r t) + e3
                        + Zp.inzmod (p %/ 2) ** bits_to_poly pt);
    return b = b';
  }
}.
```

```
Game$_{1.5}$
4 :  $\dots$
5 :  $\mu \leftarrow\$ \{0,1\}$
6 :  $\mathbf{r} \leftarrow\$ \chi^m$
7 :  $\mathbf{e} \leftarrow\$ \chi^{m+1}$
8 :  $\mathbf{u}||v \leftarrow (\mathbf{A}||\mathbf{t})^T\mathbf{r} + \mathbf{e} + \mathbf{0}^m||\frac{p}{2}\mu$
9 :
10 : $\mu' \leftarrow \mathcal{A}(\mathbf{u}, v)$
11 : return $\mu = \mu'$
```

```
Game$_2$
4 :  $\dots$
5 :  $\mu \leftarrow\$ \{0,1\}$
6 :
7 :  $\mathbf{e} \leftarrow\$ R^m$
8 :  $\mathbf{u}||v \leftarrow \mathbf{e} + \mathbf{0}^m|| \sum_{i=0}^{n-1} \mu_i\mathbf{x}^i$
9 :
10 : $\mu' \leftarrow \mathcal{A}(\mathbf{u}, v)$
11 : return $\mu = \mu'$
```

Figure 8: Using DMLWE to hop to a game where the ciphertext is uniformly random.

Once the ciphertext is uniformly random it is simple to show that the probability of winning the game is 1/2.

More formally an adversary that can distinguish between the IND-CPA game and game 1 can be used to construct an adversary which solves the DMLWE problem with equal or better advantage. This allows us to bound the advantage of an adversary against the IND-CPA game by the sum of the advantage of the adversary against game 1 and a related adversary against the DMLWE problem. A similar argument works for game 1.5 and game 2, but with a higher dimensional DMLWE problem instance. EasyCrypt handles these arguments very well, though you need to go into the details of how to construct the adversary against the DMLWE problem from the IND-CPA adversary.

```
lemma red_guess_ll: islossless RedLWEAdv.guess.
proof.
proc; call (:true); 1: apply Ag_ll.
auto => /=; conseq (:true) => />; 1: by rewrite dvector_ll dnoise_ll.
rcondf 4 => />; auto; 1: (call Ac_valid; by auto => /> result size_m0 size_m1 []).
call Ac_ll; auto => />.
apply dbool_ll.
qed.
```

In EasyCrypt we construct an explicit adversary, `RedLWEAdv`, that uses an adversary against the distinguishing game, `A` to construct an adversary against the DMLWE problem. As in the paper proof the adversary constructs an instance of the distinguishing game by using the DMLWE instance as part of the public key. We also show that this adversary is lossless. This is needed because otherwise the success probability of an adversary with low advantage might not be close to $\frac{1}{2}$ since the probability of success and failure would not have to add up to 1.

To simplify the proof we explicitly write up the IND-CPA game as `G0` with some simple reorderings and substitutions. We also prove that this is equivalent to the IND-CPA game, but this is omitted here.

```
module G0 = {
  proc main() : bool = {
    var pk,sk,mA,t,s,r,u,v,e1,e2,e3;
    var m0,m1,pt,c,b,b';
    var x: bool;

    (* Key generation *)
    s  <$ dvector dnoise m;
    e1 <$ dvector dnoise m;
    mA <$ dmatrix duni m m;
    t  <- mA *^ s + e1;
    sk <- s;
    pk <- (mA, t);

    (* Adversary chooses two messages *)
    (m0, m1) <@ A.choose(pk);

    (* Randomly encrypt one message *)
    b <$ {0,1};
    pt <- if b then m1 else m0;
    if (size pt <> n) {
      x <$ dnull;
    }
    r <$ dvector dnoise m;
    e2 <$ dvector dnoise m;
    e3 <$ dnoise;

    u <- r ^* mA + e2;
    v <- (dotp r t) + e3 +
      Zp.inzmod (p %/ 2) ** bits_to_poly pt;
    c <- (u,v);

    (* Adversary makes a guess *)
    b' <@ A.guess(c);

    return (b = b');
  }
}.
```

```
module G1(A:PKElwe.Adversary) = {
  proc main() : bool = {
    var pk, mA, t, r, u, v, e2, e3;
    var m0,m1,c,pt,b,b';
    var x: bool;

    (* Key generation /w random keys *)


    mA <$ dmatrix duni m m;
    t  <$ dvector duni m;

    pk <- (mA, t);

    (* Adversary chooses two messages *)
    (m0, m1) <@ A.choose(pk);

    (* Randomly encrypt one message *)
    b <$ {0,1};
    pt <- if b then m1 else m0;
    if (size pt <> n) {
      x <$ dnull;
    }
    r <$ dvector dnoise m;
    e2 <$ dvector dnoise m;
    e3 <$ dnoise;

    u <- r ^* mA + e2;
    v <- (dotp r t) + e3 +
      Zp.inzmod (p %/ 2) ** bits_to_poly pt;
    c <- (u,v);

    (* Adversary makes a guess *)
    b' <@ A.guess(c);

    return (b = b');
  }
}.
```

Just as the paper proof, in the EasyCrypt proof we replace the public key with a uniformly random value. We can also get rid of a few variables that are now unused.

```
lemma G0_LWEadv0_equiv &m: Pr[G0.main() @ &m : res] =
  Pr[MLWE0(RedLWEAdv).main(dvector dnoise m,m,m) @ &m : res].
proof.
byequiv (_: ={glob A} /\ arg{2} = (dvector dnoise m,m,m) ==> _) => //.
proc;inline*.
swap{1} 3 -2.
seq 3 3: (#pre /\ ={mA, s} /\ e1{1} = e{2}); 1: auto.
wp; call(_:true); auto.
sp; seq 1 1: (#pre /\ ={m0, m1}); 1: (call (: true); auto => />).
seq 1 1: (#pre /\ b{1} = b0{2}); 1: auto.
sp; if; auto.
qed.

lemma G1_LWEadv1_equiv &m : Pr[G1(A).main() @ &m : res] =
  Pr[MLWE1(RedLWEAdv).main(dvector dnoise m,m,m) @ &m : res].
proof.
byequiv (_: ={glob A} /\ arg{2} = (dvector dnoise m,m,m) ==> _) => //.
proc; inline*; wp.
call(_:true); 1: auto.
seq 6 7 : (={glob A, mA, t, m0, m1, pt} /\ b{1} = b0{2} /\ mA{1} = mA0{2}); first last.
- if; auto.
wp; rnd.
call(_:true); auto.
qed.
```

We prove that the result of the IND-CPA game has the same distribution as the real MLWE game when the second has adversary `RedLWEAdv` parameterized by the adversary to the IND-CPA game. We prove the same relation between game 1 and the fake MLWE game. Since the MLWE game is to distinguish between the fake and real game this allows us to bound the advantage of the adversary distinguising between the IND-CPA game and game 1 by the advantage of `RedLWEAdv` against the MLWE game. We will use this in the final proof.

```
module LiftLWEAdv: AdvMLWE = {
  proc guess(mAt_tr, rmAt: zpvector) : bool = {
    var mAt, mA, t, m0, m1, b, b', rmA, rt, pt;
    var x: bool;
    mAt <- trmx mAt_tr;
    mA <- subm mAt 0 (rows mAt) 0 (cols mAt -1);
    t  <- col mAt (cols mAt -1);
    (m0, m1) <@ A.choose(mA, t);
    b <$ {0,1};
    pt <- if b then m1 else m0;
    if (size pt <> n) {
      x <$ dnull;
    }
    rmA <- subv rmAt 0 (size rmAt -1);
    rt <- rmAt.[size rmAt -1];
    b'    <@ A.guess(rmA, rt + Zp.inzmod (p %/ 2) ** bits_to_poly pt);
    return b = b';
  }
}.

lemma lift_guess_ll: islossless LiftLWEAdv.guess.
proof.
proc; call (:true); 1: apply Ag_ll.
auto => /=; sp.
rcondf 4 => />; auto.
- call Ac_valid.
  by auto => /> result size_m0 size_m1 [].
call Ac_ll; auto => />.
apply dbool_ll.
qed.
```

Similarly to the first jump, the jump between game 1 and 2 requires us to come up with an explicit adversary. This time the adversary is against a higher dimensional MLWE instance with $m + 1$ columns instead of m. We also prove that this adversary is lossless.

```
module G1(A:PKElwe.Adversary) = {
  proc main() : bool = {
    var pk, mA, t, r, u, v, e2, e3;
    var m0, m1, c, pt, b, b';
    var x: bool;

    (* Key generation *)
    mA <$ dmatrix duni m m;
    t  <$ dvector duni m;
    pk <- (mA, t);



    (* Adversary chooses two messages *)
    (m0, m1) <@ A.choose(pk);

    (* Randomly encrypt one message *)
    b <$ {0,1};
    pt <- if b then m1 else m0;
    if (size pt <> n) {
      x <$ dnull;
    }
    r <$ dvector dnoise m;
    e2 <$ dvector dnoise m;
    e3 <$ dnoise;

    u <- r ^* mA + e2;
    v <- (dotp r t) + e3 +
      Zp.inzmod (p %/ 2) ** bits_to_poly pt;
    c <- (u,v);

    (* Adversary makes a guess *)
    b' <@ A.guess(c);

    return (b = b');
  }
}.
```

```
module G2(A:PKElwe.Adversary) = {
  proc main() : bool = {
    var pk, mA, mAt_tr, mAt, t, u, u', v, pt;
    var m0, m1, b, b';
    var x: bool;

    (* Key generation *)
    mAt_tr <$ dmatrix duni (m+1) m;
    mAt <- trmx mAt_tr;
    mA <- subm mAt 0 (rows mAt) 0 (cols mAt -1
    t <- col mAt (cols mAt -1);
    pk <- (mA, t);

    (* Adversary chooses two messages *)
    (m0, m1) <@ A.choose(pk);

    (* Output random ciphertext *)
    u'  <$ dvector duni (m+1);
    u <- subv u' 0 (size u' -1);
    v <- u'.[size u' -1];





    (* Adversary makes a guess *)
    b' <@ A.guess(u, v);
    b <$ {0,1};
    pt <- if b then m1 else m0;
    if (size pt <> n) {
      x <$ dnull;
    }

    return b = b';
  }
}.
```

In G2 there is one notable difference between the paper version and the EasyCrypt version. At the end we still select a random message supplied by the adversary. We need this because the adversary might give us a message with the wrong length, in which case the left side would abort. We select the message after the adversary makes their guess to emphasize that the adversary has to guess without knowledge of the selection.

It turns out that the difficult part is the hop from game 1 to game 1.5. There were two obstacles. Firstly before this work EasyCrypt did not have support for working with non-square matrices, nevermind concatenation and interactions between matrices of different sizes. To address this we have extended and generalized the EasyCrypt matrix library to include concatenation and to use dynamically sized matrices. The second obstacle is showing that sampling a vector and an element is the same as sampling a longer vector and similarly that sampling a matrix and a vector is the same as sampling a wider matrix.

```
module VecSampler1 = {
  proc main() = {
    var e2, e3;
    e2 <$ dvector dnoise m;
    e3 <$ dnoise;
    return (e2, e3);
  }
}.

module VecSampler2 = {
  proc main() = {
    var e;
    e <$ dvector dnoise (m+1);
    return e;
  }
}.

equiv vec_sampler_equiv: VecSampler1.main ~ VecSampler2.main: true ==>
  (res.`1{1} || vectc 1 res.`2{1}) = res{2} /\
    (size res{2} = (m + 1)).
proof. ··· qed.
```

The results are trivial, but the EasyCrypt proofs are very long and technical. While EasyCrypt in many cases helps you to reason about distibutions indirectly this is not possible in this case. We have to reason about the probabilities directly instead, which means we need detailed proofs about the behavior of our distributions and the process of sampling vectors and matrices.

As with the jump from the IND-CPA game to game 1 we show that the left game in the jump, in this case game 1, has results with the same distribution as the real MLWE game with adversary `LiftLWEAdv` and that the right game,

game 2, has the same distribution as the fake MLWE game with adversary `LiftLWEAdv`. These proofs are significantly longer than the corresponding proofs for the previous jump. Firstly we have to concatenate and split apart vectors and matrices. Secondly we have to deal with the transpose introduced by `LiftLWEAdv`. The first proofs also have to use the sampling lemmas from earlier.

```
lemma G2_half &m :
    Pr[G2(A).main() @ &m : res] = 1%r/2%r.
proof.
byphoare => //.
proc.
rcondf 13.
- wp; rnd.
  call(_: true); wp.
  rnd.
  call(_: true ==> size res.`1 = n /\ size res.`2 = n).
  apply Ac_valid.
  auto => /> _ _ [m0 m1] /= size_m0 size_m1 _ _ [] _ //=.
wp; rnd (pred1 b').
call(Ag_ll).
auto.
call(Ac_ll).
auto => />.
smt(dmatrix_ll dvector_ll duni_ll dbool1E).
qed.
```

We also show that all adversaries have probability $\frac{1}{2}$ of winning game 2. With all the game hops and the advantage against the last game in place we can prove our final statement.

```
lemma LWE_bound_advantage &mem:
  Pr[CPA(LWEpke, A).main() @ &mem : res] <=  1%r/2%r
    + 2%r * `|Pr[DMLWE( RedLWEAdv).main(dvector dnoise m,m  ,m) @ &mem : res] - 1%r/2%
    + 2%r * `|Pr[DMLWE(LiftLWEAdv).main(dvector dnoise m,m+1,m) @ &mem : res] - 1%r/2%
proof.
...
qed.
```

We bound the advantage of an adversary against the cryptosystem by $\frac{1}{2}$ plus the advantages of `RedLWEAdv` and `LiftLWEAdv` against their MLWE problems. This proves the reduction. If we can break the cryptosystem with significant advantage we can solve at least one of the MLWE problems with significant advantage.

For this reduction to be useful we need to know that the two derived adversaries are polynomially bounded in the runtime of the original adversary. During the writing of this thesis it became possible to reason about run-time in EasyCrypt, but it still has lacking support in the standard library. It is more complicated than one might think since EasyCrypt allows for near-arbitrary mathematical expressions, not just computable functions. In the future it might be reasonable to use the new run-time features to prove polynomial bounds, but for now we have settled on having the reader inspect the reductions.

5 Conclusion

While the cryptographic proofs contained in this work have minimal value to the research community at large considering similar, more advanced developments by large groups such as the formal verification of SABER [13] KYBER [3] and Dilithium [6], we did contribute a matrix library that has already benefitted the research comunity in the formal verification of Dilithium and can continue to see use in the EasyCrypt standard library. We also contributed several issues to the EasyCrypt issue tracker, such as the soundness bug detailed in Appendix A

References

[1] https://github.com/EasyCrypt/easycrypt-doc.

[2] Carmine Abate et al. “SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq”. In: *CSF 2021: IEEE 34st Computer Security Foundations Symposium*. Ed. by Ralf Küsters and Dave Naumann. Virtual Conference: IEEE Computer Society Press, June 2021, pp. 1–15. DOI: `10.1109/CSF51468.2021.00048`.

[3] José Bacelar Almeida et al. *Formally verifying Kyber Episode IV: Implementation Correctness*. Cryptology ePrint Archive, Paper 2023/215. `https://eprint.iacr.org/2023/215`. 2023. URL: `https://eprint.iacr.org/2023/215`.

[4] José Bacelar Almeida et al. “The Last Mile: High-Assurance and High-Speed Cryptographic Implementations”. In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 965–982. DOI: `10.1109/SP40000.2020.00028`.

[5] Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. “Definable Quotients in Type Theory”. In: ().

[6] Manuel Barbosa et al. *Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium*. Cryptology ePrint Archive, Paper 2023/246. https://eprint.iacr.org/2023/246. 2023. URL: https://eprint.iacr.org/2023/246.

[7] Gilles Barthe. *An introduction to relational program verification*. 2020.

[8] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. “CryptHOL: Game-Based Proofs in Higher-Order Logic”. In: *Journal of Cryptology* 33.2 (Apr. 2020), pp. 494–566. DOI: 10.1007/s00145-019-09341-z.

[9] Bruno Blanchet. “A Computationally Sound Mechanized Prover for Security Protocols”. In: *2006 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 2006, pp. 140–154. DOI: 10.1109/SP.2006.1.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *ITCS 2012: 3rd Innovations in Theoretical Computer Science*. Ed. by Shafi Goldwasser. Cambridge, MA, USA: Association for Computing Machinery, Jan. 2012, pp. 309–325. DOI: 10.1145/2090236.2090262.

[11] Davide Castelvecchi. “Mathematicians Welcome Computer-Assisted Proof in ‘Grand Unification’ Theory”. In: *Nature* 595.7865 (7865 June 18, 2021), pp. 18–19. DOI: 10.1038/d41586-021-01627-2. URL: https://www.nature.com/articles/d41586-021-01627-2 (visited on 03/28/2023).

[12] Georges Gonthier. “Formal Proof—The Four- Color Theorem”. In: 2008.

[13] Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. *Formal Verification of Saber’s Public-Key Encryption Scheme in EasyCrypt*. Cryptology ePrint Archive, Report 2022/351. https://ia.cr/2022/351. 2022.

[14] *Intel Takes $475 Million Charge To Replace Flawed Pentium; Net Declines*. AP NEWS. URL: https://apnews.com/article/853864f8f1a74457adf39ab272 (visited on 03/28/2023).

[15] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[16] Vadim Lyubashevsky. *Survey.Pdf*. Google Docs. URL: https://drive.google.com/file/d/1JTdW5ryznp-dUBBjN12QbvWz9R41NDGU/view?usp=embed_facebook (visited on 03/29/2023).

[17] Steven Obua. “Proving Bounds for Real Linear Programs in Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 227–244. ISBN: 978-3-540-31820-0.

[18] Adam Petcher and Greg Morrisett. “The Foundational Cryptography Framework”. In: *Principles of Security and Trust*. Ed. by Riccardo Focardi and Andrew Myers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 53–72. ISBN: 978-3-662-46666-7.

[19] Stanislas Polu et al. *Formal Mathematics Statement Curriculum Learning*. Feb. 2, 2022. DOI: 10.48550/arXiv.2202.01344. arXiv: arXiv:2202.01344. URL: http://arxiv.org/abs/2202.01344 (visited on 03/28/2023). preprint.

[20] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *37th Annual ACM Symposium on Theory of Computing*. Ed. by Harold N. Gabow and Ronald Fagin. Baltimore, MA, USA: ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603.

[21] D.I. Rich. “The evolution of systemverilog”. In: *IEEE Design & Test of Computers* 20.4 (2003), pp. 82–84. DOI: 10.1109/MDT.2003.1214355.

[22] “The Lean mathematical library”. In: *CoRR* abs/1910.09336 (2019). arXiv: 1910.09336. URL: http://arxiv.org/abs/1910.09336.

[23] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283.

A Soundness Bug

Although using formal verification can increase assurances and prevent human error in proofs they are not infallible. Even with a small trusted kernel, which EasyCrypt does not have, bugs can break the soundness of the proof assistant, allowing proofs of false statements. Often these soundness bugs cause a contradiction, but even if not they should still be taken seriously.

During this work we found a soundness bug in EasyCrypt. The bug we found has to do with while loops and can occur during rejection sampling.

Rejection sampling is a way to shrink the support of a sampling algorithm without changing the relative probability of getting any element in the support. In general it works as follows:

```
$RS(D, P)$
$x \leftarrow\!\$\, D$
while $\neg P(x)$
    $x \leftarrow\!\$\, D$
endwhile
return $x$
```

We have distibution or sampling algorithm D that we want to shrink the support of and a predicate P that is satisfied by the parts of the support we want to keep. One application of this is to use a uniform distibution on bitstrings of a fixed length to get a uniform distibution on a set of a different size, say the integers modulo some prime.

There are a number of ways to ensure that a while loop terminates with probability 1. One simple technique is to bound the probability of exiting the loop from below by a positive constant. If the probability of exiting is bounded below by p then the probability of termination is bounded below by $1-\prod_{i=1}^{\infty}(1-p)$, which is 1 when p is non-zero. EasyCrypt has a `while` tactic that uses this technique. We discovered that the tactic did not ensure that the bound was constant during the execution of the while loop. By lowering the bound fast enough every iteration we can get a positive probability for staying in the while loop indefinitely.

```
$RS(D, P)$
$n \leftarrow 1$
$x \leftarrow\!\$\, D_n$
while $\neg P(x, n)$
    $n \leftarrow n + 1$
    $x \leftarrow\!\$\, D_n$
endwhile
return $x$
```

We change the distibution to depend on a variable n that is incremented every iteration. Let p_n be the probability of exiting the loop at step n. The probability of termination is then $T = 1 - \prod_{i=1}^{\infty}(1 - p_n)$. Setting $p_n = \frac{1}{2^n+2}$

then gives $T = \frac{1}{2}$. We use the distibution $\{1, \ldots, 2^n + 2\}$ and the predicate $x = 2^n + 2$ in the EasyCrypt code and prove that the resulting procedure is lossless (terminates with probability 1), which is false.

The bug was quickly fixed, so running the code requires an older version of EasyCrypt from before the fix (https://github.com/EasyCrypt/easycrypt/pull/262). The demonstrative code we wrote is displayed below. It proves that a procedure that terminates with probability $\frac{1}{2}$ by inspection is lossless (terminates with probability 1).

```
require import AllCore DInterval List.
import StdOrder.IntOrder.Domain.
import StdOrder.IntOrder.

module RS = {
  proc sample() = {
    var n, x, found;
    n <- 1;
    found <- false;
    x <- 1;
    while (! found) {
      x <$ [1..(2^n+2)];
      if (x = 2^n+2) {
        found <- true;
      } else {
        n <- n + 1;
      }
    }
    return x;
  }
}.
lemma rejection_ll: islossless RS.sample.
proof.
proc.
sp.
seq 0: (x <= 2^n+2 /\ 1 <= n /\ (found <=> 2^n+2 = x)) => //.
auto => />. by rewrite expr1.
while (1 <= n) (if found then 0 else 1) 1 (1%r/(2^n+2)%r) => //. smt().
move => ih.
seq 2: (x <= 2^n+2 /\ 1 <= n /\ (found <=> 2^n+2 = x)) => //.
seq 1: (2^n+2 = x) (1%r/(2^n+2)%r) 1%r ((2^n+1)%r/(2^n+2)%r) 1%r
 (x <= 2^n+2 /\ 1 <= n /\ ((2^n+2 = x) \/ ! found)) => //.
auto => /> &hr _ ge1_n _ x.
```

```
rewrite supp_dinter /#.
rnd.
skip => /> &hr.
have ->:  ((=) (2^n{hr}+2)) = pred1 (2^n{hr}+2) by apply fun_ext => /#.
rewrite dinter1E => />. smt(exprn_ege1).
rcondt 1 => //.
by auto.
rnd.
skip => /> &hr.
rewrite dinterE /= size_filter => x_bound ge1_n.
rewrite (range_cat (2^n{hr}+2)) => //. smt(exprn_ege1). smt().
rewrite count_cat.
rewrite all_count_in.
rewrite allP => x.
rewrite mem_range /= /#.
rewrite count_pred0_eq_in => x.
rewrite mem_range /= /#.
rewrite size_range /=.
smt(exprn_ege1).
rcondf 1 => //.
auto => /> &hr. smt().
auto => /> &hr x_bound ge1_n eq_or_not_bound neq.
rewrite exprS. smt().
smt(exprn_ege1).
move => /> &hr _ ge1_n _.
have inj_mul: (injective (( *) (2^n{hr} + 2)%r)).
apply RField.mulfI. smt(exprn_ege1).
rewrite /injective in inj_mul.
apply (inj_mul (inv (2 ^ n{hr} + 2)%r + (2 ^ n{hr} + 1)%r / (2 ^ n{hr} + 2)%r) 1%r) =>
rewrite RField.mulrDr RField.divrr. smt(exprn_ege1).
rewrite RField.mulrA (RField.mulrC (_ + 2)%r) -RField.mulrA RField.divrr.
smt(exprn_ege1).
by rewrite !fromintD /=.
seq 1: true 1%r 0%r 0%r 0%r (x <= 2^n+2 /\ 1 <= n /\ !found) => //.
auto => /> &hr _ ge2_n _ x.
rewrite supp_dinter /#.
if; wp.
hoare => /=.
auto => />.
hoare => /=.
auto => /> &hr x_bound ge1_n _ _.
smt(exprS exprn_ege1).
```

```
auto => /> &hr ge2_n _.
rewrite weight_dinter /=.
split. smt(exprn_ege1).
move => _ x supp_dinter /#.
split.
smt(exprn_ege1).
move => z.
seq 1: (2^n+2 = x /\ z = 1) (1%r/(2^n+2)%r) 1%r ((2^n+1)%r/(2^n+2)%r) 0%r
        (x <= 2^n+2 /\ 1 <= n /\ ((2^n+2 = x) \/ ! found)) => //.
rnd.
skip => /> &hr ge1_n _ x.
by rewrite supp_dinter.
rnd.
skip => /> &hr ge1_n _.
have ->:  ((=) (2^n{hr}+2)) = pred1 (2^n{hr}+2) by apply fun_ext => /#.
rewrite dinter1E. smt(exprn_ege1).
rcondt 1 => //.
auto.
hoare => /=.
skip => />.
by rewrite expr1.
qed.
```

B Using EasyCrypt to check the proofs

Our work is tightly integrated with the Matrix library, which at the time of writing is not included in an EasyCrypt release. This means that it is necessary to compile from source to check the proofs. We are also not committing to keeping the work up to date with the EasyCrypt development branch. The latest tested commit is the commit that merged the matrix library into the EasyCrypt standard library, `a5b9b8791f8463317484f9731e26b416e8505dc4` (`https://github.com/EasyCrypt/easycrypt/pull/267`).

To check the proofs with EasyCrypt follow the EasyCrypt installation instructions in the readme (use the README in the PR commit) to install and configure the dependencies. After installing the dependencies you need to install EasyCrypt itself from source. Clone the PR commit and run `make` to build a binary from source. Run the produced binary in the command line with the file you want to check as an argument to check the file.

The artifacts can be found at https://github.com/oskgo/master-artifacts and also includes the library for quotiented polynomial rings, which we used and was written for the SABER proof [13], for convenience.